System-Specialized and Hybrid Approaches to Network Packet Classification

DISSERTATION

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von

Sven Hager

Präsidentin der Humboldt-Universität zu Berlin
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät
Prof. Dr. Elmar Kulke

1. Gutachter: Prof. Dr. Björn Scheuermann
2. Gutachter: Prof. Dr. Nils Aschenbruck
3. Gutachter: Prof. Dr. Klaus Wehrle

Tag der Einreichung: 16.01.2020
Tag der Disputation: 19.08.2020

System-Specialized and Hybrid Approaches to Network Packet Classification

Sven Hager

Abstract

Packet classification is a core functionality of a wide variety of network systems and services, such as firewalls, routers, and SDN switches. For many of these systems, throughput is of paramount importance. Further important system traits are dynamic updateability and high expressiveness in terms of rule set semantics. However, the combination of several of these properties turns packet classification into a hard problem in practice.

This work focuses on the design of classification systems and algorithms that combine at least two of the abovementioned characteristics. To this end, the concepts of hybrid systems and system specialization are employed to obtain efficient approaches to the packet classification problem in three different domains: generic classification algorithms, rule set transformation, and hardware-centric architectures.

The contributions in the domain of generic classification algorithms are the Jit Vector Search and the SFL (The Small, the Fast, and the Lazy) classification system. Jit Vector Search improves upon existing techniques by specializing the utilized search data structure on the installed rule set and by exploiting SIMD capabilities of the underlying CPU, which results in near-optimal classification performance at only slightly increased preprocessing times. In contrast, the SFL system is a hybrid approach that combines a fast classification algorithm with a lightweight update buffer to allow for high classification and update performance in dynamic environments.

With respect to rule set transformation, the RuleBender technique is proposed, which encodes decision tree structures into rule sets of practically used firewalls with jump semantics. That way, the throughput of these systems can be improved by an order of magnitude, while maintaining complex matching semantics.

Finally, the MPFC (Massively Parallel Firewall Circuits) approach is proposed, which translates a given rule set into a tailor-made matching circuit that can be implemented on an FPGA. The generated circuits are highly optimized and thus significantly smaller than those of generic matchers. To mitigate MPFC's biggest drawback, namely the long preprocessing times, the hybrid Consul approach is devised, which combines the tailor-made MPFC circuits with a generic matcher to allow for dynamic rule set updates.

Zusammenfassung

Paketklassifikation ist eine Kernfunktionalität vieler Netzwerksysteme und -dienste, wie zum Beispiel Firewalls, Router und SDN-Switche. Für viele dieser Systeme ist Durchsatz von höchster Bedeutung. Weitere wichtige Eigenschaften sind dynamische Aktualisierbarkeit und hohe Ausdrucksfähigkeit bezüglich der Regelsatzsemantik. Die Kombination dieser Eigenschaften macht Paketklassifikation zu einem schwierigen Problem in der Praxis.

Diese Arbeit befasst sich mit dem Design von Klassifikationssystemen und -algorithmen, welche mindestens zwei dieser Eigenschaften vereinen. Es werden hybride Systeme und das Konzept der Systemspezialisierung verwendet, um effiziente Ansätze zum Paketklassifikationsproblem in drei Bereichen zu erarbeiten: generische Klassifikationsalgorithmen, Regelsatztransformation und hardwarebasierte Architekturen.

Die Beiträge im Bereich der generischen Klassifikationsalgorithmen sind Jit Vector Search und das SFL (The Small, the Fast, and the Lazy)-Klassifikationssystem. Jit Vector Search verbessert existierende Techniken durch Spezialisierung der Suchdatenstruktur auf den installierten Regelsatz und durch Nutzung von SIMD-Fähigkeiten der zugrundeliegenden CPU, was in fast optimaler Klassifikationsperformanz bei kaum erhöhten Vorberechnungszeiten resultiert. Das hybride SFL-System hingegen kombiniert einen schnellen Klassifikationsalgorithmus mit einem kleinen Änderungspuffer, um sowohl hohe Klassifikations- als auch Aktualisierungsperformanz zu ermöglichen.

Bezüglich Regelsatztransformationen wird die RuleBender-Technik vorgestellt, welche Entscheidungsbäume in Regelsätze praktisch verwender Firewalls mit Sprungsemantik kodiert. Somit kann der Durchsatz dieser Systeme unter Beibehaltung komplexer Regelsatzsemantik um eine Größenordnung gesteigert werden.

Schließlich wird der MPFC (Massively Parallel Firewall Circuits)-Ansatz vorgestellt, welcher einen Regelsatz in einen auf einem FPGA implementierbaren maßgeschneiderten Matching-Schaltkreis übersetzt. Die generierten Schaltkreise sind hochoptimiert und deutlich kleiner als generische Matching-Schaltkreise. Um die langen MPFC-Vorberechnungszeiten zu entschärfen, wird der hybride Consul-Ansatz konzipiert, welcher die MPFC-Schaltkreise mit generischen Matching-Schaltkreisen kombiniert, um dynamische Regelsatzänderungen zu ermöglichen.

Danksagung

Das jahrelange Schreiben dieser Arbeit war mein persönlicher Marathonlauf. Bei diesem Lauf haben mich viele Leute begleitet, bei denen ich mich an dieser Stelle ganz herzlich bedanken möchte.

Zuallererst möchte ich Dir, Björn, Danke sagen. Du hast mir die Forschungsrichtung gewiesen, hattest immer ein offenes Ohr für meine Sorgen und Bedenken, und standest mir beim Publizieren zur Seite. Darüber hinaus hast Du mir enormen Freiraum beim Forschen gewährt und mir die Möglichkeit eröffnet, selber nach neuen Herausforderungen zu suchen. Vielen Dank, dass Du auf mich gewartet hast! Natürlich möchte ich mich auch ganz generell bei den Gutachtern dieser Arbeit für ihre Zeit, ihr Feedback und ihre Geduld bedanken.

Die Arbeit auf dem TI-Flur habe ich genossen, nicht zuletzt wegen meiner Kollegen und den vielen Studenten, mit denen ich zusammenarbeiten durfte. Flo, Du hast mich darauf aufmerksam gemacht, dass TikZ der beste Weg zum Erstellen schicker Bilder ist! Samuel und Wladik, ich bin froh, dass ich Euch bei Euren Abschlussarbeiten begleiten durfte, die Euch letztendlich zu meinen Kollegen gemacht haben. Stefan, Sebastian und Patrik, Ihr habt mit mir zusammen viele lustige Endlosrekursionen durchlebt und überwunden, dafür vielen Dank! Auch allen anderen Abschlussarbeitern, mit denen ich arbeiten durfte, möchte ich sagen: Danke für die gute Zusammenarbeit! Steffen, Dir möchte ich für Deine immerwährende Hilfsbereitschaft danken. Frank, Martin und Markus: Ihr standet mir immer zur Seite bei Fragen zu LUTs, Flip-Flops und Timingfehlern zur Seite, dafür möchte ich mich bei Euch bedanken.

Holger, ohne Dich wäre diese ganze Zeit eine andere gewesen. Mit wem hätte ich sonst die ganzen *eigentlich wichtigen* Fragen klären können? Wer hätte mir sonst erklärt, wie man TikZ-Akzente programmatisch setzt, Erlenmeyerkolben mit ordentlichen Reflektionen zeichnet und Icons von geknickten Papierblättern hochgradig parametrisierbar erzeugt? Und wer hätte die ganzen *offensichtlichen* Bugs in dreidimensionalen Plots entdeckt? Danke, dass Du mein Freund bist!

Auch möchte ich mich herzlich bei meinen Kollegen der genua GmbH sowie der genua GmbH selbst bedanken: Andreas, danke für die exzellente Zusammenarbeit im HARDFIRE-Projekt, dafür, dass Du mich zu genua geholt hast und für die Fahrt im Einhornboot! Andreas und Alexander, vielen Dank, dass Ihr mir die Möglichkeit gewährt habt, weiter an dieser Arbeit zu schreiben und sie auch tatsächlich einmal abzugeben. Sebastian und Moritz, vielen Dank für Euer Feedback zu dieser Arbeit. Claas, danke, dass Du mit Andreas und mir mitgeforscht hast und immer für gute Gespräche und Diskussionen zu haben warst. Michael, vielen Dank, dass Du bei mir Deine Masterarbeit geschrieben, mit uns zusammen im HARDFIRE-Projekt geforscht, mir beim Be- und Entwässern meines Rechners geholfen und mir die *absolute und uneingeschränkte Überlegenheit* der C++-Sprache bewiesen hast.

Schließlich möchte ich mich bei meiner Familie bedanken, die mich all die Jahre unterstützt hat. Mutter, Vater, danke, dass Ihr mir das Studium ermöglicht habt, was es mir erlaubt hat, diese Arbeit zu schreiben. Auch bei Dir, Rolf, bedanke ich mich: Du hast geholfen, mir den Weg zu weisen und standest mir immer mit Rat zur Seite. Anastasia und Cassandra, meine kleinen Töchter: Ihr habt die Endphase dieser Arbeit interessant gestaltet und mir einen *sehr* guten Grund gegeben, zum Ende zu kommen. Dafür danke ich Euch!

Alina, Du bist den ganzen Weg mit mir gegangen. Du hast mit mir studiert, bist mit mir nach Berlin gekommen, hast mich geheiratet, hast mit mir die auch die dunklen Stunden durchlebt. *Und Du bist bei mir geblieben. Du hast mich aufgebaut und mir gezeigt, dass die Nacht vor der Dämmerung am finstersten ist. Dafür werde ich Dir immer dankbar sein!*

In Verbundenheit,
Sven

Contents

Prolog

1 Introduction

1.1 Motivation and Challenges

The discrimination of network packets plays a vital role in network infrastructures, as virtually any computing device connected to a network must be able to distinguish between different types of network traffic: personal computers or workstations often employ software firewalls such as `iptables` [167] for threat protection, Quality of Service (QoS) routers inspect multiple packet header fields in order to route and prioritize traffic [30], Software-defined Network (SDN) switches forward packets based on information installed by a controller [73], and Intrusion Detection Systems (IDSs) execute Deep Packet Inspection (DPI) in order to detect malware in packet payloads [74, 139]. Despite their widely different tasks and purposes, the abovementioned systems share the commonality that their packet discrimination process is guided by a *rule set*. A rule set is a list of different traffic classes to which network packets are mapped, such that the system can decide the further treatment of the processed packets. For example, a firewall needs to decide whether to accept or drop a packet, while a router requires knowledge about the forwarding interface for an incoming packet. The process of mapping packets to the corresponding class(es) in the rule set is known as *packet classification* [61], and systems which conduct this task are called *packet classification systems*.

Although packet classification is an old problem that has been extensively researched since the 1990s [31, 62, 63, 76, 122, 127, 128, 146], it is still considered a difficult problem from a practical perspective [83] due to a variety of requirements: first, packet classification systems should aim for a high classification performance. If packets cannot be classified at least as fast as they enter the classification system, the system might cause a bottleneck in the network, when used as a middlebox or forwarding device. Second, packet classification systems should be able to handle growing rule set sizes, as rule sets tend to grow rather than to shrink in practice [137]. This is especially important for systems that employ many fine-grained rules, such as SDN switches. Third, updates to the rule set should be integrated quickly into the matching process. If this is not the case, the system might become unresponsive or does not implement the desired matching behaviour. Finally, some systems may require more complex matching

semantics than typical subnet or range checks on certain packet header fields. For example, many software-based firewalls are able to track connection states or even execute arbitrary user-defined programs when inspecting incoming packets [165, 166, 172]. The difficulty of packet classification arises from the combination of two or more of the abovementioned requirements: building a system that handles thousands of rules with complex matching semantics while simultaneously executing frequent rule set updates at runtime is significantly harder than to build a system with just a few simple static rules.

1.2 Contributions

As packet classification is of high practical relevance for current network infrastructures, there exists a plethora of approaches and applications with respect to specific classification aspects, both in terms of practical solutions as well as mostly theoretical techniques. Among the well-known practical applications are firewalls, routers, (SDN) switches, Internet Protocol Security (IPsec) appliances, Intrusion Detection Systems, and packet filtering Virtual Machines (VMs) [94, 99]. Existing approaches devised by the research community as well as practical applications to the packet classification problem can roughly be grouped into the following three categories: *algorithmic classification techniques*, *rule set transformation*, and *hardware-centric approaches*. In this work, we[1] contribute to each of the three categories by proposing approaches that employ specialization techniques for performance enhancements or combine two diametrically opposed ideas to hybrid systems which provide the best traits of both worlds.

1.2.1 Algorithmic Classification Techniques

Algorithmic classification techniques focus on the traversal of generic search data structures that are computed when a rule set is installed or updated in a classification system. When a packet enters the classification system, the algorithm traverses the precomputed data structure in order to detect which rules match the packet. To achieve high classification performance, most existing algorithms trade updateability for high lookup speed, i. e., they rely on sophisticated search data structures that require high preprocessing times and large memory footprints which in turn can quickly be traversed [31, 58, 62, 63, 76, 107, 115, 122, 137]. However, practical classification systems, such as `iptables` [167] or Open vSwitch [105], instead use lightweight algorithms, such as Linear Search or Tuple

[1] In this work, first person plural pronouns such as “we” or “our” are used to refer to the single author of this thesis in order to improve readability.

Space Search [127], which provide good update performance, but fall short in terms of classification performance.

Here, we address this issue of diametrically opposed performance traits of classification algorithms by proposing two novel approaches. First, we present an extension to the existing Bit Vector Search (BV) scheme [76] which we call *Jit Vector Search*. Although the BV approach does not provide a canonical way to implement dynamic rule set updates, it still provides reasonably fast update performance at high lookup speed. By extending the BV algorithm with dynamically generated machine code specific for the installed rule set and Single Instruction Multiple Data (SIMD) instructions, we can significantly increase its matching performance at moderate increases in preprocessing costs. That way, we are close to the fastest existing classification algorithm *Recursive Flow Classification (RFC)* [62] in terms of lookup speed, but avoid all of RFC's drawbacks, namely huge preprocessing times and memory explosions. *Parts of the algorithmic ideas behind Jit Vector Search are published in a co-authored work [1] (second author) and in [17] (first author).*

Second, we propose a generic hybrid classification system design we call *The Small, the Fast, and the Lazy (SFL)*. The SFL system implements an update wrapper around an existing fast classification algorithm, such as RFC. Incoming rule set updates are stored in an update buffer, which is propagated lazily into the fast search data structure of the wrapped algorithm. We then use the update buffer to adjust potentially outdated classification results from the fast search data structure. As a result, incoming rule set updates immediately take effect and lead to significantly lower deterioration in classification performance, as the search data structure is not recomputed for every single rule set update. *The main idea of the SFL approach is published in [10] (first author).*

1.2.2 Rule Set Transformation

Rule set transformation refers to the process of altering a rule set towards a specific goal, typically while preserving the rule set's semantics. Existing rule set transformation techniques aiming to improve system performance often target the reduction of the rule set size [49, 52, 71, 84, 86, 88] to reduce the memory footprint of the underlying search data structure. On Central Processing Unit (CPU)-based classification systems, this can result in a faster traversal of the search data structure, while on special purpose matching hardware, memory modules can be utilized more effectively. However, existing general purpose multidimensional rule set reduction techniques, such as [49, 71, 84, 88], are not particularly well suited to improve the performance of linear-search-based software firewalls such

as `iptables` [167], `nftables` [168], or `ipfw` [165]. First, reducing the size of a rule set that is searched linearly still results in a system that exhibits slow linear lookup speed. Second, many of these systems provide complex match semantics, which is not supported by existing rule set transformators.

In order to improve the performance of the abovementioned firewall systems, we propose the rule set transformation technique *RuleBender*. Instead of aiming to shrink a rule set's size, RuleBender deliberately increases it by encoding decision tree search data structures [63, 107] into the rule set itself. To this end, RuleBender exploits a widely implemented feature within firewalls, namely the concept of *jump actions*. A jump action redirects the matching flow during a classification to a different position in the rule set, which is the key feature that enables the decision tree encoding. The RuleBender technique can improve the matching performance of existing systems by an order of magnitude, and furthermore supports rule sets with a variety of complex checks. Moreover, RuleBender can be combined with existing minimization-based approaches to a hybrid transformator, which can further improve the achievable throughput, provided that the transformation target does not heavily rely on complex checks. *The presented RuleBender approach is published in [11] (first author), earlier ideas that finally led to RuleBender are published in [13] (first author) and [12] (first author).*

1.2.3 Hardware-centric Approaches

Hardware-centric approaches rely on dedicated matching hardware, such as Application-specific Integrated Circuits (ASICs) like Ternary Content-addressable Memorys (TCAMs) [66] or Field-programmable Gate Arrays (FPGAs). They deliver line rate classification by taking advantage of SIMD circuitry, pipelined parallelism, and massive memory bandwidths. Especially FPGAs provide attractive implementation platforms for high speed classification systems, as they can be easily and arbitrarily often (re)configured and avoid the expensive manufacturing process required for function-fixed ASICs. Accordingly, many classification approaches have been proposed and evaluated on FPGAs, most of which exhibit a strict separation between the memories that store the search data and the actual matching circuitry [56, 68, 69, 109, 125].

In this work, we argue that this separation of search data structure and match logic gives away a fundamental advantage that FPGAs have over ASICs, namely their reconfigurability. We therefore present the *Massively Parallel Firewall Circuit (MPFC)* approach, which translates a specified rule set into a corresponding tailor-made matching circuit that is used to configure the matching semantics into the FPGA. Due to the fact that the MPFC circuits solely consist of matching

logic specifically crafted for a single rule set, they can be heavily optimized. As such, the MPFC circuits require significantly fewer hardware resources than a comparable on an FPGA implemented generic matching circuit that relies on configuration memories for the search data structure. *The idea behind specialized matcher generation is published in [14] (first author), [15] (first author), [9] (first author), and [8] (first author).*

However, as every rule set change with the MPFC approach requires a costly rebuild of the FPGA configuration, we propose to use a hybrid matching pipeline. This hybrid approach contains hardwired and thus highly optimized MPFC matching circuits, and additionally utilizes an existing generic matching approach with support for dynamic rule set changes at runtime. The resulting circuitry combines the advantages of both approaches, because its hardware resource footprint is still lower than a purely generic approach, while it provides viable update capabilities. Furthermore, the proposed hybrid matching circuit protocol is not limited to the usage of MPFC, but can also be applied to combine multiple generic matchers with different traits and strengths. *The approach of hybrid matching pipelines is published in a co-authored work [2] (second author).*

1.3 Outline

Before we dive into the details of the abovementioned techniques, we begin with a formal introduction to the packet classification and rule set transformation problems in Chapter 2 in order to pave the ground for the remainder of this work. We then turn our attention to algorithmic packet classification techniques, including both related work and the proposed Jit Vector Search and SFL approaches, which are presented in Part I. Subsequently, we move on to the proposed RuleBender algorithm, which is described in Part II alongside existing related techniques. Next, we describe the concept of specialized matching circuit generation, the idea of hybrid matching pipelines, and existing generic approaches in Part III. Finally, we conclude this work with the Epilog.

2 Problem Statement

In this chapter we introduce the main challenges addressed in this thesis, namely the *Packet Classification Problem* and the *Rule Set Transformation Problem*. We begin with an informal introduction of packet classification in Section 2.1 in order to establish a general understanding of the core concepts we are concerned with. In Section 2.2, we formally introduce the notion of packets, rule sets, and system states to pave the ground for the subsequently given problem statements. We define the Geometric Packet Classification Problem in Section 2.3, as it represents the core of the abovementioned problems. Subsequently, we move on to the Complex Packet Classification Problem in Section 2.4, and finally focus on the Rule Set Transformation Problem in Section 2.5.

2.1 Packet Classification: Use Cases and Relevance

In general, packet classification refers to the process of distinguishing between incoming network packets on behalf of certain data fields within these packets [61]. The specific set of fields taken into consideration depends on the regarded use case: for example, link layer switches and Internet Protocol (IP) routers typically perform single-field packet classification by forwarding packets through the inspection of either the destination Media Access Control (MAC) or IP address, respectively. In contrast, network firewalls and OpenFlow switches [97] often use multiple header fields for packet discrimination. Often regarded fields are MAC and IP addresses, protocol numbers, or fields specific to transport layer protocols like Transmission Control Protocol (TCP) or User Datagram Protocol (UDP), such as ports. Also, applications like intrusion detection/prevention systems [104, 113] or application level gateways may additionally inspect the message payload carried by the packet under discrimination. Finally, some classification systems

Our definition of the packet classification problem is inspired by the classic description given in [61], which we further extend by the definition of complex checks based on the author's previous (co-authored) works [3, 6]. Finally, the notation of the Rule Set Transformation Problem has its roots in the publications by Liu et al. [84, 88] with respect to rule set size reduction as well as in previous first-authored publications [11, 13].

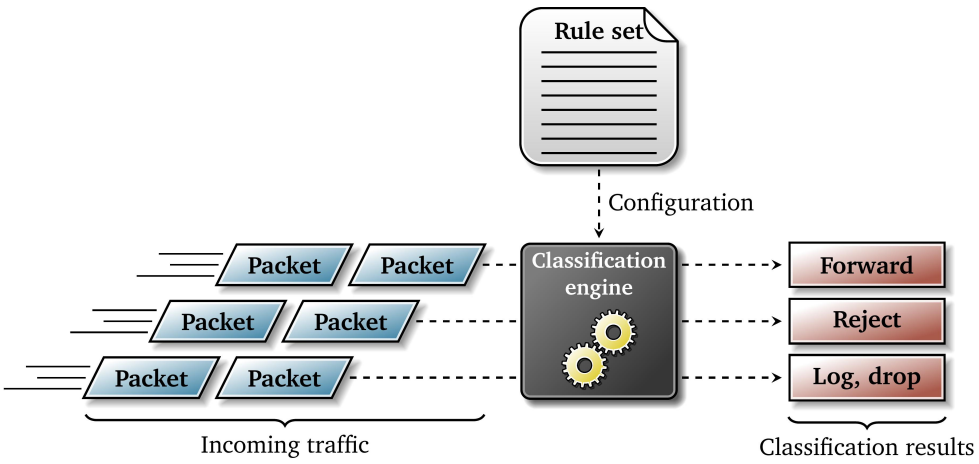


Fig. 2.1: Classification systems dispatch incoming traffic based on an installed rule set.

may use some form of state stored on the classification machine as an additional input criterion to process incoming packets. For instance, a firewall often uses state tables to track network connections.

Although the abovementioned systems serve different purposes in IP networks, e. g., forwarding, access control, or attack prevention, they have in common that the packet distinction process is guided by a previously installed *policy* or *rule set*. These rule sets, often specified by a system administrator or also composed programmatically, map incoming network packets into distinguished *traffic classes* or *flows*. These flows are typically treated differently by the classification system, as depicted in Figure 2.1. To this end, a rule set specifies a sequence of *filters* or *rules* that each formally define a corresponding traffic class by narrowing down the set of allowed packet types. Furthermore, each rule provides an *action* to be executed for every packet that falls into the defined traffic class. For each incoming packet, the task of the underlying classification engine is to identify the packet's traffic class based on the installed rule set. Subsequently, the action assigned to this specific class through corresponding rules in the rule set is executed.

For example, a classical perimeter firewall rule set, as illustrated in Table 2.1, will in most settings distinguish between at least two major traffic classes: legal and illegal packets [155, 59, 164]. Legal packets, such as Hypertext Transfer Protocol (HTTP) or Domain Name System (DNS) requests from inside of a company and the corresponding server responses, are forwarded across network boundaries, which is implemented by the rules R_1 to R_4. In contrast, packets considered illegal are either silently dropped or rejected by the firewall, as specified by the entirely wildcarded rule R_5. Note that we assumed in the example that rules are prioritized, and that a rule R_i's priority decreases with an increasing index i. Accordingly, for two rules R_i and R_j, R_i is more highly prioritized than R_j if $i < j$.

Nr. / Priority	Source subnet	Destination subnet	Source port	Destination port	Transport protocol	Complex check	Action
R_1	54.17.102.0/24	*	*	80	TCP	Add connection	Accept
R_2	*	54.17.102.0/24	80	*	TCP	Check connection	Accept
R_3	54.17.102.0/24	8.8.8.8/32	*	53	UDP	-	Accept
R_4	8.8.8.8/32	54.17.102.0/24	53	*	UDP	-	Accept
R_5	*	*	*	*	*	-	Reject

Tab. 2.1: An example firewall rule set ('*' denotes a wildcard).

Therefore, in the example rule set, every packet that is not explicitly accepted within the first four rules is ultimately rejected by the firewall.

Having seen a practical example of packet classification, we formally introduce the different problems as well as the notation used throughout this thesis in the remainder of this section.

2.2 Fundamental Definitions

2.2.1 Packets, Packet Space, and Header Space

For a given Maximum Transmission Unit (MTU) [178], we define a *network packet*

$$p \in P_{\text{MTU}} \tag{2.1}$$

as a sequence of at most MTU-many bytes, with

$$P_{\text{MTU}} \tag{2.2}$$

being the *packet space* set of all possible network packets with at most MTU-many bytes. We assume that a certain portion of a packet p's byte sequence is used to represent the integer *header values* h^p within their respective *header field domains*

$$h^p := \left(h_1^p \in H_1, \ldots, h_d^p \in H_d\right), \tag{2.3}$$

while all remaining bytes form the *message payload* m^p. Thus, the packet p can be regarded as a tuple

$$p := (h^p, m^p). \tag{2.4}$$

According to the typical IP packet composition [179], we define the header field domains H_j as intervals of non-negative integers with

$$H_j \coloneqq \left[0, 2^{Y_j} - 1\right] \text{ with } Y_j \in \mathbb{N}. \tag{2.5}$$

We denote the crossproduct of the header field domains as the *header space*

$$\mathcal{H} \coloneqq H_1 \times \ldots \times H_d. \tag{2.6}$$

Furthermore, let $\mathcal{A}$ be a set of *actions*, such as ACCEPT or DROP. Additionally, we define the Boolean space $\mathbb{B}$ as

$$\mathbb{B} \coloneqq \{\text{true}, \text{false}\}. \tag{2.7}$$

2.2.2 Rules and Rule Sets

We consider a *rule set* $\mathfrak{R}$ of size n to be an ordered list of n rules $R_1, \ldots, R_n$, which we write as

$$\mathfrak{R} \coloneqq \langle R_1, \ldots, R_n \rangle. \tag{2.8}$$

For rule sets that consider network packets with at most MTU-many bytes, we define the set of all possible rule sets as

$$\mathfrak{S}_{\text{MTU}}. \tag{2.9}$$

Although we do not consider rule sets as sets in the strict mathematical sense, as we allow duplicate elements and also require the abovementioned ordering of rules, we will use some notations typically used for sets in the remainder of this work for purposes of clarity and brevity. For example, we write

$$|\mathfrak{R}| \tag{2.10}$$

to refer to rule set $\mathfrak{R}$'s size, i. e., the number of rules within $\mathfrak{R}$. Similarly, we use the notation

$$R \in \mathfrak{R} \tag{2.11}$$

to indicate that the rule R exists in $\mathfrak{R} = \langle R_1, \ldots, R_n \rangle$, i. e.,

$$\exists i \in \{1, \ldots, n\} : R_i = R. \tag{2.12}$$

We model a *rule* R as a $(d+2)$-tuple

$$R \coloneqq (g_1, \ldots, g_d, c, a)\,. \tag{2.13}$$

Here, R consists of d *geometric checks* g_j, a *complex check* c, and an action a. In essence, the checks g_j and c determine whether a rule R matches an incoming packet. The exact definition of checks as well as their match semantics are defined in the subsequent sections 2.3 and 2.4, respectively. A rule's action a specifies the operations that are carried out by the classification system in case of a matching rule. For example, common actions like DROP or ACCEPT will either discard or accept an incoming packet, and terminate the classification process. Other actions like JUMP or LOG do not assign a verdict to the packet, but instead alter the classification system's state by redirecting the classification flow or writing log files. Accordingly, we define the set of all possible actions that may be executed by a specific classification system by

$$\mathcal{A} \coloneqq \mathcal{A}_{\text{term}} \cup \mathcal{A}_{\text{nonterm}}. \tag{2.14}$$

We define $\mathcal{A}_{\text{term}}$ as a set of *terminal actions* that terminate the classification process and assign a verdict to the packet, such as DROP or ACCEPT. Also, we define $\mathcal{A}_{\text{nonterm}}$ as a set of *non-terminal actions* which do not end the classification process and also do not assign a verdict to the packet, such as LOG or JUMP. Instead, non-terminal actions may cause side effects or state changes within the classification system, e. g., in order to redirect the classification flow or to create log entries.

2.3 The Geometric Packet Classification Problem

In this section we introduce the Geometric Packet Classification Problem (GPCP) based on the notion of actions, rules, and geometric checks. Generally, geometric, or stateless, packet classification allows the filtering of network packets on the basis of selection predicates [101], which are applied to certain parts of the packet header. We refer to these predicates by the term *geometric check*.

The geometric checks g_j are functions

$$g_j : H_j \to \mathbb{B} \tag{2.15}$$

that verify whether the jth header value h_j^p of a packet p fulfills a certain matching condition. More specifically, we assume that every g_j is either an *equality check*

$$g_j(h_j) \coloneqq (h_j = x) \quad \text{with} \quad x \in H_j, \tag{2.16}$$

a *prefix check*

$$g_j(h_j) \coloneqq (h_j \in x/y) \quad \text{with} \quad x \in H_j \quad \text{and} \quad y \in [0, Y_j]\,, \tag{2.17}$$

or a *range check*

$$g_j(h_j) \coloneqq (h_j \in [x, y]) \quad \text{with} \quad x, y \in H_j. \tag{2.18}$$

The idea behind equality and range checks is straightforward: an incoming header value h is equality-tested either against a single value x, or range-tested against an interval $[x, y]$. Prefix checks, on the other hand, test whether a header value h's y leftmost bits are equal to the y leftmost bits of a value x. For example, in the header space domain $H = [0, 2^3 - 1]$, the prefix check $g = (6/2) = (110_2/2)$ verifies whether the first two bits of a header value are set. g can therefore be also written as the regular expression `11*` ('*' refers to `1` or `0`), which we call the *ternary representation* of the prefix check g.

We say that a geometric check g_j *matches* the header field h_j iff $g_j(h_j) = \text{true}$. Equality checks, prefix checks, and range checks are the most common tests used in packet classification systems [31, 72, 127]. While prefix checks are typically used for IP address fields, range and equality checks are most often applied on transport layer ports, protocol fields, or MAC addresses [109].

An important property of geometric checks is the fact that they can always be viewed as a hyperrectangle in a d-dimensional space, which, in our case, is the header space $\mathcal{H}$. Note that, although we defined three types of geometric checks, each geometric check can be represented as a range check and therefore as an interval. For instance, an arbitrary equality check $h_j = x$ can be represented by the range check $h_j \in [x, x]$. Likewise, an Internet Protocol Version 4 (IPv4) prefix check

$$h_j \in x/y \text{ with } x \in \left[0, 2^{32} - 1\right] \text{ and } y \in [0, 32] \tag{2.19}$$

is equivalent to the range check

$$h_j \in \left[\left\lfloor \frac{x}{2^{32-y}} \right\rfloor \cdot 2^{32-y}, \left\lfloor \frac{x}{2^{32-y}} \right\rfloor \cdot 2^{32-y} + \left(2^{32-y} - 1\right)\right]. \tag{2.20}$$

Accordingly, for a given rule $R = (g_1, \ldots, g_d, c, a) = ([x_1, y_1], \ldots, [x_d, y_d], c, a)$, we define $B(R)$ as the hyperrectangle (or *box representation*)

$$B(R) \coloneqq [x_1, y_1] \times \ldots \times [x_d, y_d]\,. \tag{2.21}$$

Thus, in geometrical terms, a rule's box representation can be considered a d-dimensional rectangle, as sketched for the three-dimensional rule $R = ([x_1, y_1],$

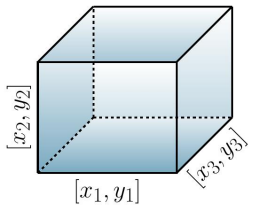


Fig. 2.2: Representing a rule with three geometric checks as a hyperrectangle.

$[x_2, y_2]$, $[y_3, y_3]$, $c, a)$ in Figure 2.2. In fact, this property is an essential ingredient for certain classification algorithms and preprocessing techniques, as we will see in Part I and Part II.

Note that, while every geometric check can be represented as a range check, not every range check can natively be represented by a prefix check. For instance, in the header space domain $H = [0, 3]$, the range check $g = (h \in [1, 3])$ has no direct prefix check equivalent. Since g matches the binary integers `01`, `10`, and `11`, there exists no prefix check that exactly matches on these numbers. Instead, two prefix checks $g' = (h \in 1/2) =$ `01` and $g'' = (h \in 2/1) =$ `1*` are required to represent the same matching behaviour as g, because g' matches only `01` and g'' matches only `10` and `11`. Because some classification approaches [114, 127] require all geometric checks within a rule set to be represented as prefix checks, we say that a rule R is in *prefix format* if every geometric check g_j is either a prefix check or an equality check.

We define the *geometric match function* γ_R for a rule R with

$$\gamma_R : \mathcal{H} \to \mathbb{B} \tag{2.22}$$

and

$$\gamma_R\left(h^p\right) \coloneqq \bigwedge_{j=1}^{d} g_j\left(h_j^p\right). \tag{2.23}$$

According to its definition, a rule R's geometric match function γ_R determines for a given packet p whether all of p's header values match the corresponding geometric checks. For a given packet p and a rule R, we call $\gamma_R\left(h^p\right)$ the *geometric match result* of R for p.

Having introduced the notion of geometric match functions, we now define the *Geometric Packet Classification Problem* in Problem 2.1. The general goal of the geometric packet classification problem is to lookup the most highly prioritized rule R_{i^*} in a rule set and extract, if such a rule exists, the corresponding terminal action a^{i^*} for an incoming packet p. If no matching rule exists, we use the *no-match symbol* ϵ to indicate this circumstance. In either case, the result of the

GEOMETRIC PACKET CLASSIFICATION PROBLEM	
Given:	· Rule set $\mathfrak{R} = \langle R_1, \ldots, R_n \rangle$ · Packet $p \in P_{\text{MTU}}$ · Terminal action set $\mathcal{A}_{\text{term}}$ · No-match symbol ϵ
Searched:	The verdict $v_p \in \mathcal{A}_{\text{term}} \cup \{\epsilon\}$
	with
	$v_p = \begin{cases} a^{i^*}, \text{ if } \exists i^* \in \{1, \ldots, n\} : \gamma_{R_{i^*}}(h^p) \wedge \\ \quad (\nexists i \in \{1, \ldots, i^* - 1\} : \gamma_{R_i}(h^p)) \\ \epsilon, \text{ otherwise} \end{cases}$

Problem 2.1: The Geometric Packet Classification Problem.

lookup is stored in the *verdict* v^p, which is used by the classification engine in order to determine the fate of the packet p. We define the function

$$f_{\text{GPCP}} : \mathfrak{S}_{\text{MTU}} \times P_{\text{MTU}} \rightarrow \mathcal{A}_{\text{term}} \cup \{\epsilon\} \tag{2.24}$$

with

$$f_{\text{GPCP}}(\mathfrak{R}, p) \coloneqq v_p. \tag{2.25}$$

to implement the semantics of Problem 2.1.

The above definition describes a *single-match* classification problem, which finds at most one matching rule. Single-match classification must not be confused with *multi-match* classification, which seeks to compute *all* matching rules, not only the most highly prioritized one [148]. In the reminder of this work, we will always refer to single-match classification, if not mentioned otherwise. Furthermore, we always assume that a rule R_i is more highly prioritized than a rule R_j if $i < j$, i. e., we assume *first match semantics*. This stands in contrast to some practically used firewall systems, such as [172], which prioritize rules with a higher index (*last match semantics*). However, our definition of the packet classification problem can still be applied to those systems, as it is always possible to convert rule sets with last match semantics to equivalent rule sets with first match semantics [149].

Note that the GPCP is based entirely on geometric checks defined within a rule set, and does neither utilize an incoming packet's payload nor a stored machine state to reach a decision. Especially, the complex checks that are defined in the rules are not used. Hence, the problem describes the classical *stateless* packet classification process [115], and as such, we permit only terminal actions in the GPCP that do not cause side effects. Also note that the GPCP is similar to the *Planar Point*

Location Problem [119], which maps a point in a plane to its enclosing polygon. However, the GPCP is only equivalent to the Planar Point Location problem if the rules' box representations do not intersect with each other.

Although the geometric packet classification problem is less expressive in terms of matching semantics than the Complex Packet Classification Problem, as described in Section 2.4, approaches to geometric packet classification are at the heart of a plethora of existing classification systems, including firewalls, Intrusion Detection Systems (IDSs)/Intrusion Prevention Systems (IPSs), and Internet Protocol Security (IPsec) gateways [177, 74, 115]. Furthermore, certain techniques to quickly solve the geometric packet classification problem can be generalized to also efficiently tackle classification on more complex rule sets, which we demonstrate in Part II.

2.4 The Complex Packet Classification Problem

There exist situations where the sole use of geometric checks is not expressive enough in order to model certain kinds of more complex filtering behaviour. An example for such a situation can be seen in rules R_1 and R_2 in Table 2.1: here, the intention is to prevent arbitrary TCP packets with source port 80 to enter the 54.17.102.0/24 subnet. Instead, the classification engine requires that a corresponding connection state entry has been created previously by an outgoing packet that matched rule R_1. Such a connection entry typically stores at least a timestamp of the last seen packet in this connection as well as the four-tuple of source and destination addresses/ports. That way, incoming packets only match rule R_2 if the connection has not expired and if they have been explicitly requested previously. Other examples for complex matching requirements are string matching within a packet's payload or load balancing on the basis of packet counters or random numbers.

Here, for a classification system running on an underlying machine configured with a rule set $\mathfrak{R}$, we denote the set of all possible *classification-relevant states* stored on the machine by Σ. Examples for classification-relevant states are connection tables, packet counters, the state of a random number generator, and generally any further information that is utilized in the classification process for a given packet using the rule set $\mathfrak{R}$. In order to also model complex checks for a rule R_i, we define R_i's complex check c^i as

$$c^i : P_{\text{MTU}} \times \Sigma \times \mathbb{B} \rightarrow \mathbb{B} \times \Sigma \tag{2.26}$$

with

$$c^i\left(p, \sigma_{i-1}^p, \gamma_{R_i}\left(h^p\right)\right) \coloneqq \left(\beta, \sigma_i^p\right). \tag{2.27}$$

and

$$\begin{array}{rcll} p & \in & P_{\text{MTU}} & : \text{ the current packet} \\ \sigma_{i-1}^p & \in & \Sigma & : \text{ the classification-relevant state } \textit{before} \text{ the} \\ & & & \text{execution of } c^i \\ \gamma_{R_i}\left(h^p\right) & \in & \mathbb{B} & : \text{ the geometric match result of } R_i \text{ for } p \\ \beta & \in & \mathbb{B} & : \text{ the match result of the complex check } c^i \\ \sigma_i^p & \in & \Sigma & : \text{ the classification-relevant state } \textit{after} \text{ the} \\ & & & \text{execution of } c^i \end{array} \tag{2.28}$$

In contrast to geometric checks, c^i may use not only a packet p's header values, but also p's payload m^p, the relevant state information σ_{i-1}^p, and the match results of $R_i's$ geometric checks $\gamma\left(h^p\right) \in \mathbb{B}$ to reach a match result. Also, a complex check may change the classification-relevant state of the classification system. We define $\sigma_{i-1}^p \in \Sigma$ as the classification-relevant state that is accessible to the classification system immediately before executing the complex check c^i of rule R_i on the packet p. Furthermore, we refer to $\sigma_i^p \in \Sigma$ as the classification-relevant state immediately after the execution of c^i. Analogously to geometric checks, we say that a complex check c^i *matches* a packet p iff $\beta =$ true with

$$c^i\left(p, \sigma_{i-1}^p, \gamma_{R_i}\left(h^p\right)\right) = \left(\beta, \sigma_i^p\right). \tag{2.29}$$

Note that σ_{i-1} does not necessarily have to differ from σ_i^p.

We now define two subclasses of complex checks that are required in the remainder of this thesis, namely *stateless complex checks* and *match-based complex checks*. A complex check c^i is referred to as a *stateless complex check* if c^i never alters the classification-relevant state. Also, we say that a complex check c^i is *match-based*, if it may only change the classification-relevant state if (1) all geometric checks evaluate to true, and (2), if the complex check itself matches the regarded packet, i. e.,

$$\sigma_{i-1} \neq \sigma_i \Rightarrow \gamma_{R_i}\left(h^p\right) \wedge \beta. \tag{2.30}$$

Examples for complex tests that can be implemented with match-based complex checks are string searches in packet payloads or connection tracking. In contrast, a complex check that uses a random number generator to randomly match is not match-based, because the generator's state changes also in the no-match case.

For a rule R_i that does not specify a complex check, we model the complex check c^i as a function c_{id} with

$$c_{\text{id}} : P_{\text{MTU}} \times \Sigma_M \times \mathbb{B} \to \mathbb{B} \times \Sigma \tag{2.31}$$

and

$$c_{\text{id}}\left(p, \sigma, \gamma_{R_i}\left(h^p\right)\right) \coloneqq (\text{true}, \sigma) \;\forall\; \left(p, \sigma, \gamma_{R_i}\left(h^p\right)\right) \in P_{\text{MTU}} \times \Sigma_M \times \mathbb{B} \tag{2.32}$$

that does not change any classification-relevant state with the complex match decision *true*.

We refer to a rule R with

$$R = (g_1, \ldots, g_d, c_{\text{id}}, a) \tag{2.33}$$

as a *geometric rule*, for which we also use the shorthand notation

$$R = (g_1, \ldots, g_d, a)\,. \tag{2.34}$$

Otherwise, R is referred to as a *complex rule*. Likewise, we call a rule set $\mathfrak{R}$ a *geometric rule set* if every rule in $\mathfrak{R}$ is a geometric rule, i. e., iff

$$\forall i \in \{1, \ldots, n\} : c_i = c_{\text{id}}. \tag{2.35}$$

Otherwise, we call $\mathfrak{R}$ a *complex rule set*.

Finally, a rule R_i's *complex match function* κ_{R_i} is defined as

$$\kappa_{R_i} : P_{\text{MTU}} \times \Sigma \to \mathbb{B} \tag{2.36}$$

with

$$\kappa_{R_i}\left(p, \sigma_{i-1}^p\right) \coloneqq \gamma_R\left(h^p\right) \wedge \beta \tag{2.37}$$

and

$$(\beta, \sigma_i^p) = c^i\left(p, \sigma_{i-1}^p, \gamma_R\left(h^p\right)\right). \tag{2.38}$$

In essence, a rule R's complex match function determines whether an incoming packet matches R with respect to R's geometric checks as well as R's complex check.

With a proper definition of complex checks, we are now able to extend the GPCP to the *Complex Packet Classification Problem (CPCP)*, as defined in Problem 2.2. The major difference between these problems are the matching semantics and the handling of classification-relevant states. In contrast to the GPCP, the CPCP

COMPLEX PACKET CLASSIFICATION PROBLEM

Given:
· Rule set $\mathfrak{R} = \langle R_1, \ldots, R_n \rangle$
· Packet $p \in P_{\text{MTU}}$
· Action set $\mathcal{A} = \mathcal{A}_{\text{term}} \cup \mathcal{A}_{\text{nonterm}}$
· No-match symbol ϵ
· Initial classification-relevant state σ_0^p

Searched:
The tuple (verdict $v^p \in \mathcal{A} \cup \{\epsilon\}$, new state σ_*^p)

with

$$v^p := \begin{cases} a^{i^*}, \text{ if } \exists i^* \in \{1, \ldots, n\} : \kappa_{R_{i^*}}\left(p, \sigma_{i^*-1}^p\right) \wedge \\ \quad a^{i^*} \in \mathcal{A}_{\text{term}} \wedge \\ \quad (\nexists i \in \{1, \ldots, i^* - 1\} : \kappa_{R_i}\left(p, \sigma_{i-1}^p\right) \\ \quad \text{with } a^i \in \mathcal{A}_{\text{term}}) \\ \epsilon, \text{ otherwise} \end{cases}$$

and

$$\sigma_*^p := \begin{cases} \sigma_{i^*}^p, \text{ if } v_p \neq \epsilon \\ \sigma_n^p, \text{ otherwise} \end{cases}$$

Problem 2.2: The Complex Packet Classification Problem.

includes the execution of complex checks as well as non-terminal actions, which may both lead to potential state changes. Note that the CPCP allows multiple matching rules, as long as only the last matching rule defines a terminal action. Analogously to the function f_{GPCP}, we define the function

$$f_{\text{CPCP}} : \mathfrak{S}_{\text{MTU}} \times P_{\text{MTU}} \times \Sigma \to (\mathcal{A} \cup \{\epsilon\}) \times \Sigma \tag{2.39}$$

with

$$f_{\text{CPCP}}\left(\mathfrak{R}, p, \sigma_0^p\right) := \left(v^p, \sigma_*^p\right). \tag{2.40}$$

to implement the semantics of Problem 2.2.

It is important to distinguish between the CPCP and the GPCP because not every classification algorithm suited for the GPCP is also able to solve the CPCP. The same holds for rule set transformation approaches: in Part II we will present a novel rule set optimization technique that, in contrast to related work, is able to not only optimize geometric rule sets, but also complex rule sets when match-

based complex checks are used. In order to ease notation in the remainder of this work, we define the *rule set decision function* $f_{\mathfrak{R}}$ as

$$f_{\mathfrak{R}}\left(p, \sigma_0^p\right) := \begin{cases} f_{\text{GPCP}}\left(\mathfrak{R}, p\right), \text{ if } \mathfrak{R} \text{ is a geometric rule set} \\ v_p, \text{ with } \left(v_p, \sigma_*^p\right) = f_{\text{CPCP}}\left(\mathfrak{R}, p, \sigma_0^p\right) \text{ if } \mathfrak{R} \text{ is a complex rule set.} \end{cases} \tag{2.41}$$

Also, we will abbreviate $f_{\mathfrak{R}}\left(p, \sigma\right)$ to $f_{\mathfrak{R}}\left(p\right)$, whenever $\sigma = \sigma_0^p$, thereby treating σ_0^p as a default parameter to σ.

2.5 The Rule Set Transformation Problem

All major classification systems, from plain packet forwarding engines over firewalls to complex IDSs, have in common that they discriminate packets on the basis of an installed rule set. In this section, we introduce the concept of *rule set transformations*, which take a specified input rule set and generate a corresponding output rule set. The need for rule set transformations can arise from a variety of reasons, which we briefly motivate before we dive into our formal definition of the Rule Set Transformation Problem. Depending on the underlying implementation platform, a classification system's performance may heavily depend on the structure and/or the size of the utilized rule set. For example, software-based systems typically provide better matching speed when fewer rules have to be tested at runtime, as the rules' checks are in many cases executed in a sequential manner [88, 115]. Even systems that do not employ a classification algorithm with sequential components often profit from smaller rule sets, as they may reduce the size of the utilized search data structure [62]. Furthermore, certain matching algorithms and/or implementation platforms require rule sets to be represented in a certain format, as the underlying search data structure may only implement specific kinds of checks. For instance, Ternary Content-addressable Memory circuits only support rule sets in prefix format [114]. Finally, from an administrative point of view, it may be useful to detect and remove redundant rules from a rule set in order to increase an administrator's understanding of the rule set [71]. This, in turn, reduces the risk of misconfigurations and potential security issues [85].

In this work, we are solely concerned with *semantics-preserving rule set transformations*, as depicted in Figure 2.3. As the name suggests, a semantics-preserving transformation of a rule set $\mathfrak{R}$ generates a rule set $\mathfrak{R}'$ with the same matching semantics. Consider a given *transformation function*

$$t : K_{\text{MTU}} \subseteq \mathfrak{S}_{\text{MTU}} \rightarrow \mathfrak{S}_{\text{MTU}} \tag{2.42}$$

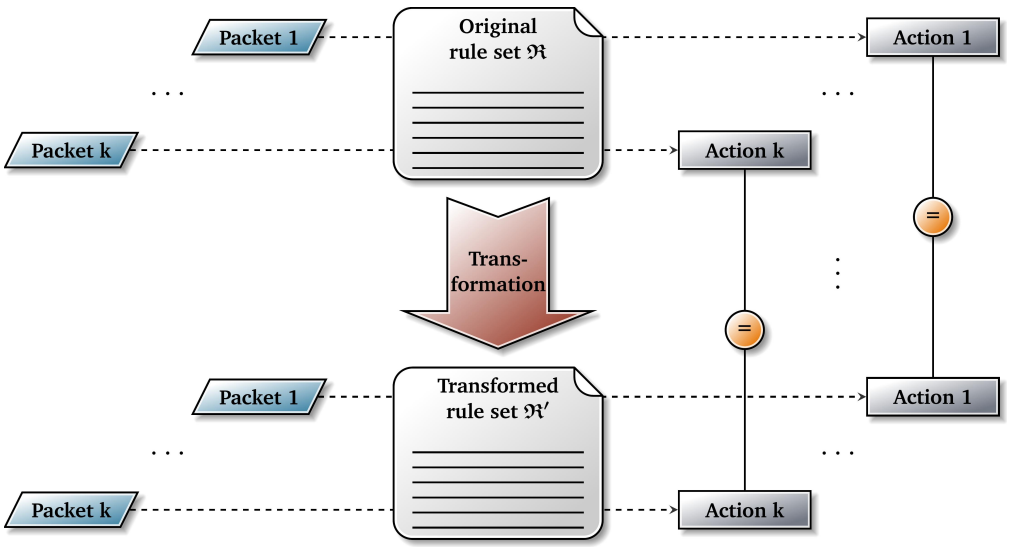


Fig. 2.3: Executing a semantics-preserving rule set transformation.

that maps an input rule set $\mathfrak{R}$ to an output rule set $\mathfrak{R}'$ with

$$t\left(\mathfrak{R}\right) = \mathfrak{R}'. \tag{2.43}$$

We say that t is a *semantics-preserving transformation function* if for any given sequence of packets $\langle p_1, \ldots, p_k \rangle \in P^k_{\text{MTU}}$ the following condition holds:

$$\forall i \in \{1, \ldots, k\} : f_{\mathfrak{R}}\left(p_i\right) = f_{t(\mathfrak{R})}\left(p_i\right). \tag{2.44}$$

As rule set transformation functions may only be able to operate on a subset $K_{\text{MTU}} \subseteq \mathfrak{S}_{\text{MTU}}$, we restrict ker t to K_{MTU}.

Of course, rule set transformations are typically executed to achieve a certain goal with respect to the input and output rule sets. For example, a rule set minimization transformation aims to compute an output rule set $\mathfrak{R}'$ from an input rule set $\mathfrak{R}$, such that $|\mathfrak{R}'| \leq |\mathfrak{R}|$. In order to model the goal of a transformation function, we define a *rule set property* $\pi_{\mathfrak{S}_{\text{MTU}}}$ as a function

$$\pi_{\mathfrak{S}_{\text{MTU}}} : \mathfrak{S}_{\text{MTU}} \times \mathfrak{S}_{\text{MTU}} \to \mathbb{B}. \tag{2.45}$$

A rule set property encodes the goal of a transformation process by mapping a tuple $(\mathfrak{R}, \mathfrak{R}')$ into the Boolean space. If $\pi_{\mathfrak{S}_{\text{MTU}}}\left(\mathfrak{R}, \mathfrak{R}'\right) = \text{true}$, the rule set transformation was executed successfully, otherwise it failed.

Using semantics-preserving transformation functions and rule set properties, we now define the *Rule Set Transformation Problem (RSTP)* in Problem 2.3. Note that, from a practical point of view, it may not always be possible to compute the verification result $\pi_{\mathfrak{S}_{\text{MTU}}}\left(\mathfrak{R}, \mathfrak{R}'\right)$. For example, when it comes to the removal of

RULE SET TRANSFORMATION PROBLEM	
Given:	· Rule set property $\pi_{\mathfrak{S}_{\text{MTU}}}$ · Subset $K_{\text{MTU}} \subseteq \mathfrak{S}_{\text{MTU}}$
Searched:	A semantics-preserving transformation function $t : K_{\text{MTU}} \rightarrow \mathfrak{S}_{\text{MTU}}$ with $\pi_{\mathfrak{S}_{\text{MTU}}}(\mathfrak{R}, t(\mathfrak{R})) = \text{true}$ $\forall\, \mathfrak{R} \in K_{\text{MTU}}$

Problem 2.3: The Rule Set Transformation Problem.

redundant rules from a rule set with arbitrary complex checks, $\pi_{\mathfrak{S}_{\text{MTU}}}(\mathfrak{R}, \mathfrak{R}')$ is not decidable [111].

An example for a rule set property used in the remainder of this thesis is the function

$$\pi_{\mathfrak{S}_{\text{MTU}}}(\mathfrak{R}, \mathfrak{R}') := |\mathfrak{R}'| \leq |\mathfrak{R}|. \tag{2.46}$$

The abovementioned property demands that the size of the transformed rule set $\mathfrak{R}'$ is smaller or equal to the size of the original rule set $\mathfrak{R}$, which is the idea behind rule set reduction techniques [49, 71, 84, 88].

Part I

Classification Algorithms

3 Introduction

Algorithmic packet classification is the most prevalent form of network packet classification in today's computing infrastructure, as it is done by virtually any computer capable of network packet processing. Most of these devices use common off-the-shelf Central Processing Units (CPUs) and Random-access Memory (RAM) components as their underlying computing hardware, and as such, rely on generic algorithmic problem solving techniques rather than on specialized hardware components for most tasks. Of course, the packet processing and classification requirements are straightforward for most systems, as they simply need to accept incoming packets and transmit outgoing packets. However, when it comes to dedicated software routers, switches, perimeter firewalls, or IDS/IPS, the performance of network packet classification is of paramount importance, as entire subnets with hundreds or thousands of machines may be bottlenecked by those systems. Such a scenario is sketched in Figure 3.1. Besides the raw packet classification performance, several other metrics, such as rule set update speed or data structure memory footprint, can, depending on the classification scenario, be of equal or even higher importance [105].

Accordingly, the research community has proposed a wide variety of classification algorithms, which range from hash- and cache-based approaches [127, 128] over multi-dimensional decision trees [63, 83, 107, 115, 122, 137, 146] to table- and decomposition-based algorithms [31, 58, 62, 76, 81, 147]. Most of these techniques mainly focus on classification performance at the cost of high memory requirements and long preprocessing times [58, 62, 63, 107, 122, 147], which can lead to unresponsive classification systems or even failed search data structure creation attempts [81, 83]. Especially the fastest existing classification algorithm

The first version of the Jit Vector Search approach, as presented in Chapter 5, was prototyped in Samuel Brack's bachelor's thesis [21], which was supervised by the author and subsequently published [1] as co-author. The (A)JV Search presented in this work is an evolution of the initial Jit Vector Search and improves it by faster lookups in small dimensions and by employing Single Instruction Multiple Data (SIMD) instructions for faster vector processing. Parts of (A)JV's dynamic code generation backend were implemented by Michael Offel, who was a student assistant in Björn Scheuermann's group.

The proposed SFL algorithm, as presented in Chapter 6, was first prototyped in Samuel Brack's master's thesis [20], which was supervised by the author, and subsequently published [10] in a first-authored publication. Large parts of SFL, as used in the evaluation, were implemented by Samuel Brack during his master's thesis. Although not directly related to the SFL approach, Wladislaw Gusew's CATE framework, which was developed in his master's thesis [22] (supervised by the author) and later published [7] as co-author, was a strong inspiration for the SFL design with respect to interface and modularity.

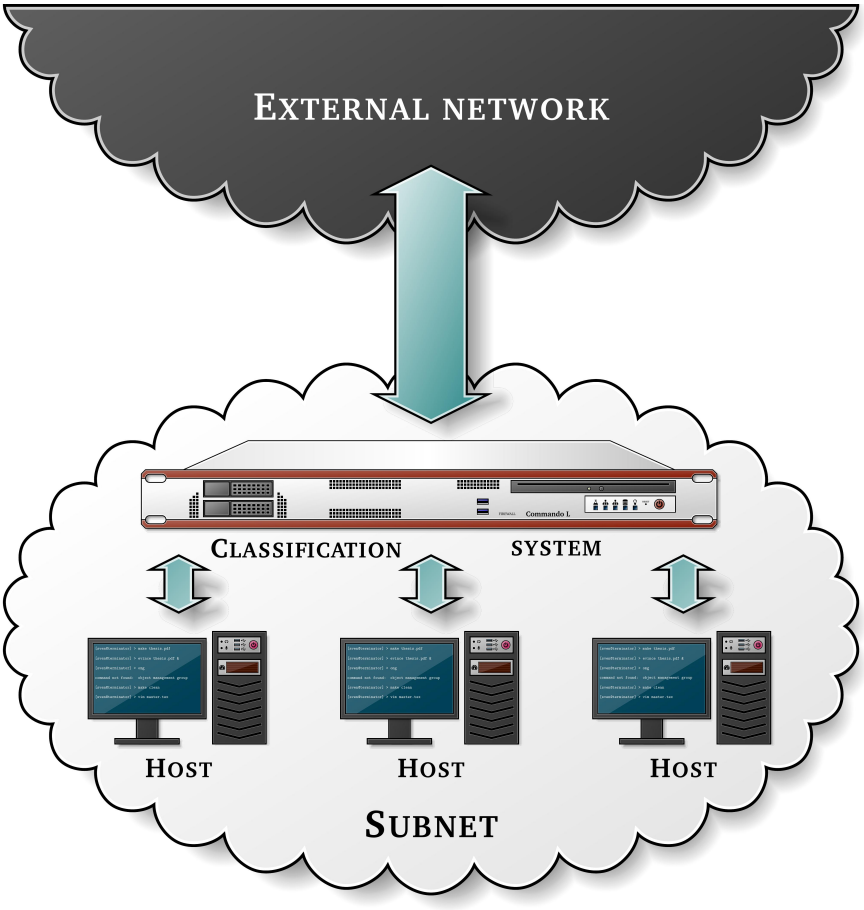


Fig. 3.1: Sketch of a subnet consisting of many host systems whose in- and outbound traffic is processed by a perimeter classification system (e. g., a firewall), which may bottleneck the traffic in case of poor classification performance. (Parts of this figure were created by the author during his regular work at genua GmbH and are used in this work with genua's permission.)

Recursive Flow Classification (RFC) [62], which classifies packets regardless of the rule set size in constant time, suffers from memory explosions and huge preprocessing times [58, 81]. On the other hand, the highly updateable and dynamic *Tuple Space Search* and *Linear Search* algorithms often do not meet line speed requirements, as the per-packet processing overhead can significantly rise with growing rule set sizes [105, 122].

In this part, our goal is to find classification approaches which provide both high classification performance as well as quick search data structure modifications. We begin by proposing an extension to the existing (Aggregated) Bit Vector Search algorithm [31, 76], that breaks up the typically strict separation between search data structure and algorithm implementation by specializing the algorithm's implementation on a specific rule set. To this end, we employ dynamically generated machine code whenever the rule set changes in order to improve the classification performance while avoiding too high preprocessing costs. Moreover, we further

augment the search algorithm by exploiting the SIMD capabilities of current CPUs to further accelerate lookups at practically no additional memory or preprocessing overhead. The resulting rule set- and CPU-specialized (A)BV Search, which we call *(Aggregated) Jit Vector Search ((A)JV)*, significantly outperforms existing approaches except for RFC, and does not suffer from any of RFC's abovementioned problems. As such, (A)JV Search is well suited for practical use cases, as it inherits the exactly predictable quadratic preprocessing duration and memory footprints from (A)BV Search, significantly accelerates lookups, and provides further improvement potential with developing SIMD capabilities of common CPUs.

Although the (A)JV algorithm provides good classification performance at comparatively short data structure preprocessing times, it still needs quadratic time for rule set updates due to its non-incremental search data structure. In fact, most existing fast classification algorithms utilize data structures that do not support incremental updates at all [62, 147] or may suffer from severe data structure deteriorations in case of dynamic updates [63, 107, 122, 137]. Especially in the case of decision tree algorithms, the initial preprocessing phase to compute the search tree(s) is well understood, but most approaches do not provide techniques for partial updates [63, 107, 122, 137]. Although one recent algorithm, namely *CutSplit* [82], seems to provide low preprocessing times, its classification performance is neither compared to RFC [62] nor to (A)BV [31, 76], which we find in our evaluation (Section 5.4) to be the fastest classification algorithms. Moreover, another publication [83] hints that CutSplit often provides lower lookup speed than the basic HiCuts [63] scheme (to which the CutSplit publication does not compare), which is significantly outperformed by RFC and (Aggregated) Bit Vector Search, as shown in Section 5.4.

On the other hand, existing dynamically updateable algorithms, such as Linear Search and Tuple Space Search [127], can process rule set changes quickly in linear time, at the cost of significantly lower lookup performance [61, 105]. Thus, we ask the question: *is it possible to design a generic classification algorithm that provides high classification performance and quick rule set updates at the same time?* In order to answer this question, we propose the *SFL classification system*, which wraps the search data structure of an existing classification algorithm to allow for delayed, or lazy, search data structure recomputations. At the same time, however, each issued rule set update is still taken into account during the classification process, such that no outdated classification results are computed. Using this technique, the SFL approach can reach peak classification performance in static classification setups, such as perimeter firewalls, and still provide incremental update capabilities, which is particularly important in dynamic environments, such as Software-defined Networks (SDNs) [105].

The remainder of this part is structured as follows: in Chapter 4, we review existing classification algorithms in detail. Subsequently, we present the (Aggregated) Jit Vector Search in Chapter 5, and then move on to the hybrid SFL approach, which we describe in Chapter 6. Finally, we summarize this part in Chapter 7.

4 Related Work

Since the late 1990s, a wide variety of different packet classification schemes have been proposed, both in generic variants for nearly arbitrary underlying hardware platforms, as well as hardware-centric approaches. In this chapter, we focus on existing generic classification algorithms, as they form the vantage point for the proposed Jit Vector Search and SFL techniques. Due to the large number of existing schemes, we mostly limit ourselves to classification approaches that are vastly present in the relevant literature [61, 131]. However, for every algorithm we discuss in detail, we will point out to existing related approaches when appropriate. Furthermore, we loosely follow Taylor's taxonomy [131] of classification approaches into *exhaustive searches*, *decomposition techniques*, *decision tree schemes*, and *tuple space search* by providing examples for each of these algorithm families.

We begin our overview with a description of *Linear Search* in Section 4.1, the most basic exhaustive search technique. Subsequently, we depict *Tuple Space Search* [127] in Section 4.2. Next, we move on to *bit vector algorithms* [31, 76] and *crossproducting approaches* [128] in Sections 4.3 and 4.4, respectively, which are representatives of decomposition techniques. Thereafter, we depict the principle of *decision trees* [63, 107] in Section 4.5. Finally, we briefly address approaches based on Virtual Machines (VMs) to network packet classification in Section 4.6.

4.1 Linear Search

The most straightforward algorithm to classify an incoming packet p is a *Linear Search* over the installed rule set $\mathfrak{R}$. As the name suggests, this technique sequentially scans the rule set $\mathfrak{R}$ until the most highly prioritized rule R_{i^*} is either found or all rules have been traversed without a successful match. This process is illustrated in Figure 4.1 for a packet p with the most highly prioritized matching rule R_3. The classification of a packet using Linear Search requires at most $\mathcal{O}(d \cdot n)$

The algorithmic techniques described in this chapter exclusively refer to existing related work by other authors. These techniques are depicted in the author's words to provide an overview and understanding of existing state-of-the-art, against which the author's work can be compared.

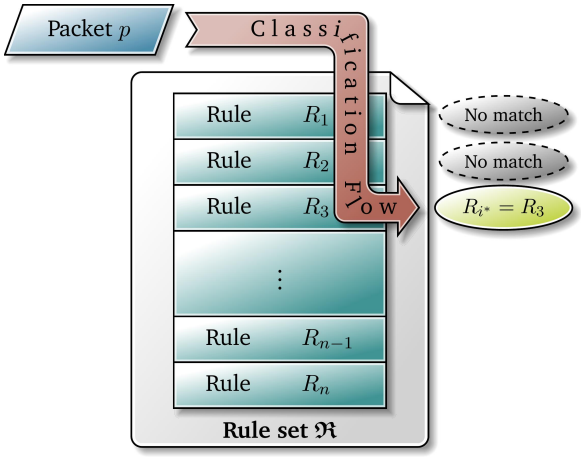


Fig. 4.1: Classifying a packet using *Linear Search*.

Classification operation	Data structure creation	Data structure update	Memory requirements
$\mathcal{O}(d \cdot n)$	$\mathcal{O}(d \cdot n)$	$\mathcal{O}(n)$	$\mathcal{O}(d \cdot n)$

n: number of rules d: number of fields

Tab. 4.1: Linear Search performance characteristics.

operations, because each field of every traversed rule must be tested against the corresponding packet header in the worst case.

Of all existing classification algorithms, Linear Search provides the smallest memory footprint for the applied search data structure, which is typically an array or a linked list. Therefore, the storage requirements are linear in the number of rules times the number of fields per rule. Due to the simple search data structure, Linear Search also provides excellent data structure creation and update performance, because rule insertions or deletions can be directly mapped to the underlying list structure. Specifically, rule insertions or deletions require one write or deletion operation at the specified position in the list structure. If the list structure is an array, this position can be accessed in constant time, but all array slots behind the update position must be shifted by one to the left (for deletions) or to the right (for insertions). If instead a linear list is used, the update position must first be detected by traversing the list, because the rule items are not stored in a coherent block of memory. Accordingly, an update operation also requires linear time in the number of rules. An overview of the operation and memory complexities of Linear Search is given in Table 4.1.

The Linear Search approach is a widely deployed classification algorithm, as it is used in numerous open source packet filters and firewalls, such as Linux' `netfilter/iptables` [167], FreeBSD's `ipfw` [165] and `ipf` [175], as well as in OpenBSD's `pf` [172]. Due to its simplicity, it is easy to implement and to extend

in terms of packet matching capabilities, as the large number of over 80 existing `netfilter/iptables` extensions demonstrates [166]. However, its modularity and its good memory/update performance stand in sharp contrast to its comparatively slow classification speed: when configured with more than just a few rules, a classification system's throughput will likely suffer throughput break-ins under high packet load [63, 13, 76]. Therefore, more advanced classification algorithms trade higher memory footprints and longer update durations for considerably faster search operations, as we will see in the remainder of this chapter.

4.2 Tuple Space Search

The *Tuple Space Search* technique, originally proposed in [127], is a dynamic classification algorithm that builds upon the good lookup performance of hash tables in order to process incoming network packets. Tuple Space Search builds upon the observation that the rules $R_1, \ldots, R_n$ in a d-dimensional rule set $\mathfrak{R}$ can be partitioned into m equivalence classes $C_1, \ldots, C_m$ based on the rules' structural properties. These equivalence classes are effectively sub rule sets and can, by exploiting the structure of all rules within one specific equivalence class, quickly be searched independently. One important aspect behind this approach is that, in typical rule sets, the number of equivalence classes is much smaller than the number of rules, i. e., $m \ll n$ [127].

In order to determine the equivalence classes, a prefix rule

$$R_i = \left(h_1 \in x_1^i/y_1^i, \ldots, h_d \in x_d^i/y_d^i, a^i\right) \tag{4.1}$$

is mapped to the d-tuple $t_i = (y_1^i, \ldots, y_d^i)$. Two prefix rules R_i and R_j are said to be in the same equivalence class, if their corresponding tuples t_i and t_j are equal, i. e., $t_i = t_j$. However, if a rule is not in prefix format, it must first be converted into a set of prefix rules before the tuple mapping can take place. Unfortunately, this procedure increases preprocessing time and will blow up the size of the stored rule set in many cases, as there often does not exist a one-to-one mapping between a non-prefix rule to a prefix rule. Alternatively, the non-prefix rules can be kept in a separate list which is searched linearly during the classification process. This approach avoids the increase of the rule set size, but can result in poor classification performance, if there are too many non-prefix rules. In the remainder of this section, however, we will assume that every rule R_i is in prefix format in order to focus on the actual classification algorithm.

After the tuples have been extracted from every rule, a hash map M_t is created for every *distinct* tuple t. Subsequently, every rule R_i is inserted into the hash map

M_{t_i} that corresponds to the rule's tuple t_i. Here, a rule R_i's hash key is computed using the relevant leftmost y_j^i bits of the net address x_j^i from the rule's subnet checks, which we denote by $x_j^i|_{y_j^i}$. For a given hash function f, the hash key $k(R_i)$ for the prefix rule R_i can be obtained by hashing the concatenated relevant parts of the subnet checks, i. e.,

$$k(R_i) = f(x_1^i|_{y_1^i} x_2^i|_{y_2^i} \dots x_d^i|_{y_d^i}). \tag{4.2}$$

After all $(k(R_i), R_i)$ pairs have been stored in the hash maps, an incoming packet p can be classified by linearly probing the m hash maps. In order to search for rules that match the packet p within a hash map $M_{t=(y_1,\dots,y_d)}$, a hash key $k(p)$ is constructed using p's header fields analogously to the rule hashing procedure, i. e.,

$$k(p) = f(h_1^p|_{y_1} h_2^p|_{y_2} \dots h_d^p|_{y_d}). \tag{4.3}$$

Subsequently, $k(p)$ is used to locate all rules in M_t that could potentially match p, which are those rules that also hash to $k(p)$. Of these rules, the most highly prioritized matching rule R_{M_t} is extracted. Note that, in general, this process must be repeated for all m hash maps, since the rule hashing process does not preserve the initial rule ordering. Finally, after all hash maps have been traversed, the overall most highly prioritized matching rule R_{i^*} is determined from the rules that are most highly prioritized in their respective hash maps. Both the computation of the hash maps from a two-dimensional initial rule set $\mathfrak{R}$ and the classification operation for a packet p are illustrated in Figure 4.2.

As the previously in Section 4.1 discussed Linear Search, Tuple Space Search has a memory footprint linear in the number of rules, since every rule is stored in a single hash map, plus a small bookkeeping overhead introduced by the hash maps. However, if we assume amortized $\mathcal{O}(1)$ hash map lookup performance, probing m hash maps is still significantly faster than linearly scanning n rules (if $m \ll n$). Moreover, Tuple Space Search allows for dynamic rule set updates that do not require a rebuild of the entire search data structure, as rule insertions or deletions can be performed through the corresponding hash map operations. These performance characteristics are summarized in Table 4.2.

A practical implementation of the Tuple Space Search algorithm can be found in the Open vSwitch [105]. According to the authors of [105], Tuple Space Search has been selected over faster classification algorithms due to its small storage requirements as well as its capability for quick rule set updates. Furthermore, although Tuple Space Search has been primarily designed for the usage on general purpose CPU systems [127], it has successfully been implemented and evaluated

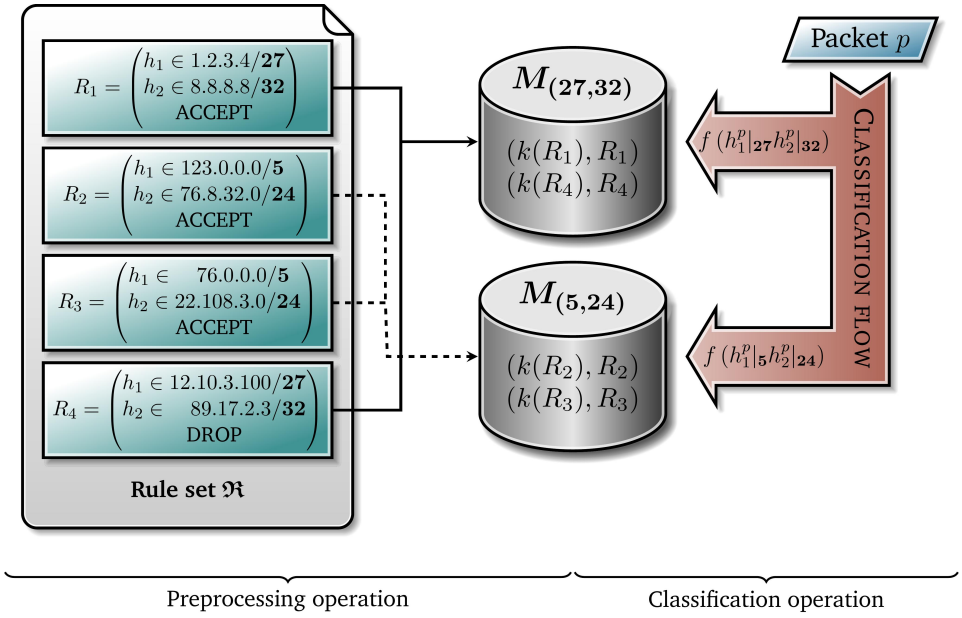


Fig. 4.2: Sketch for Tuple Space Search preprocessing and classification operations.

Classification operation	Data structure creation	Data structure update	Memory requirements
$\mathcal{O}(m)$	$\mathcal{O}(d \cdot n)$	$\mathcal{O}(1)$	$\mathcal{O}(d \cdot n + m)$

n: number of rules d: number of fields m: number of distinct tuples

Tab. 4.2: (Amortized) Tuple Space Search performance characteristics.

on a GPU platform [138]. That way, all hash maps can be queried in parallel, which allows to further accelerate the lookup operation.

4.3 Bit Vector Algorithms

The classification algorithms we have covered until this point, namely Linear Search and Tuple Space Search, rely on lightweight search data structures that require little effort to initially construct and to maintain. However, these approaches do not take the geometric representation in a d-dimensional coordinate system of rules into account, which provides several opportunities to further boost the classification performance at runtime, typically at the cost of increased preprocessing times and larger memory footprints. In contrast, the *bit vector algorithms* covered in this section exploit the d-dimensional rectangle representation of rules in order to generate d independent search data structures that can be implemented efficiently both in software and hardware.

4.3.1 Lucent Bit Vector Search

The original *Bit Vector Search* proposed by Lakshman and Stiliadis [76], which is also referred to as the *Lucent Bit Vector Scheme* [31] or simply the *Lucent Scheme* [127], is a decompositional classification algorithm, in the sense that it decomposes the original d-dimensional classification problem into d one-dimensional search problems that can quickly be solved independently. Once the d partial solutions for the one-dimensional problems are computed, they can be combined to obtain the desired overall solution, which is the index of the most highly prioritized matching rule. In the remainder of this section, we refer to the Bit Vector Search algorithm by the abbreviation *BV*.

In its preprocessing phase, BV constructs d one-dimensional search data structures from the geometric view of the specified rule set $\mathfrak{R}$. In order to visualize the bit vector preprocessing, we use the two-dimensional rule set shown in Table 4.3 as a running example throughout this section. Each rule R_i in $\mathfrak{R}$ is regarded as a d-dimensional hyperrectangle $B(R_i) = [X_1^i, Y_1^i] \times \ldots \times [X_d^i, Y_d^i]$ in the bounding box $B(\mathcal{H})$ of the header space $\mathcal{H}$, i. e., $B(R_i) \subseteq B(\mathcal{H})$. In the first step of the preprocessing phase, the endpoints X_j^i and Y_j^i of each rule R_i are projected onto the jth axis of the bounding box $B(\mathcal{H})$. Thereby, for all $1 \leq j \leq d$, the jth axis is partitioned into α_j disjoint intervals I_k^j, with $1 \leq \alpha_j \leq 2n + 1$. In the next step, a *bit vector* V_k^j of length n is assigned to each interval I_k^j. Each bit b_i in the bit vector V_k^j indicates whether an for incoming packet p, whose jth header value h_j^p may fall into the interval I_k^j, matches rule R_i in the jth dimension. Accordingly, V_k^j's ith bit is set iff the interval I_k^j intersects with rule R_i's jth check interval $\left[X_j^i, Y_j^i\right]$, i. e.,

$$\forall\, i \in \{1, \ldots, n\} : V_k^j[i] = \begin{cases} 1, \text{ if } I_k^j \cap \left[X_j^i, Y_j^i\right] \neq \varnothing \\ 0, \text{ otherwise.} \end{cases} \tag{4.4}$$

This procedure is sketched in Figure 4.3 for the rule set from Table 4.3.

After the intervals I_k^j and the corresponding bit vectors V_k^j have been computed in the algorithm's preprocessing phase, the classification of an incoming packet

Nr. / Priority	Field F_1	Field F_2	Action
R_1	$[\ 2,\ 5]$	$[\ 2,\ 9]$	a^1
R_2	$[12, 13]$	$[\ 4,\ 5]$	a^2
R_3	$[\ 4,\ 9]$	$[\ 8, 13]$	a^3
R_4	$[\ 8, 15]$	$[10, 11]$	a^4

Tab. 4.3: A two-dimensional example rule set over $\mathcal{H} = [0, 15]^2$.

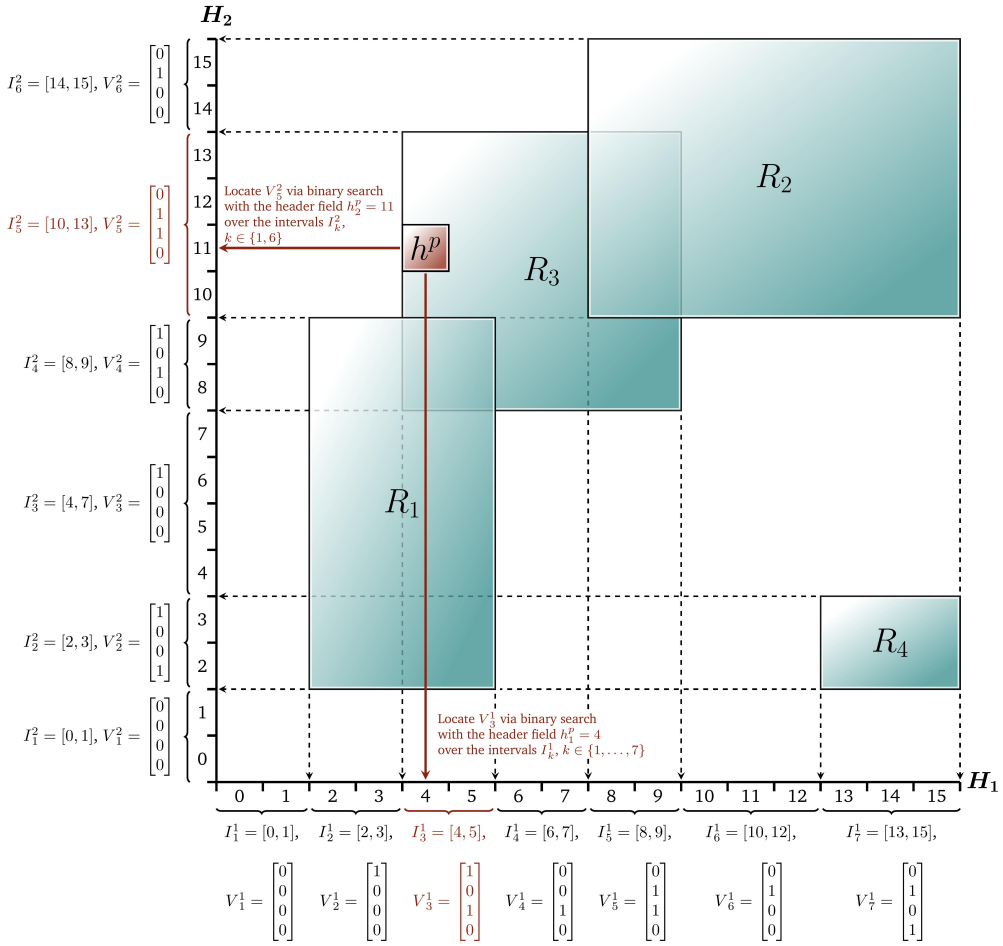


Fig. 4.3: Sketch of the Bit Vector Search data structure generated for the geometric representation of the two-dimensional rule set $\mathfrak{R}$ over the header space $\mathcal{H} = H_1 \times H_2 = [0,15]^2$ from Table 4.3. The packet p with the header $h^p = (4,11)$ is used to illustrate the bit vector retrieval.

p is executed in two consecutive steps. First, each of the packet p's header fields h_j^p is used to retrieve the bit vector V^j that belongs to the interval I^j containing h_j^p, as sketched in Figure 4.3 for the packet header $h^p = (4,11)$. Note that there always exists exactly one such interval, since for every header field dimension, the entire header field domain H_j is partitioned into mutually disjoint intervals. Subsequently, the retrieved bit vectors, which each represent the matching information for a single dimension, are bitwise ANDed in order to obtain a final result vector V_{res} for p, i. e.,

$$\forall\, i \in \{1,\ldots,n\} : V_{\text{res}}[i] = \bigwedge_{j=1}^{d} V^j[i]\,. \tag{4.5}$$

Accordingly, each bit $V_{\text{res}}[i]$ is set iff p matches the rule R_i in every regarded dimension. Therefore, finding the most highly prioritized matching rule R_{i^*} translates to finding the index i^* of the first set bit $V_{\text{res}}[i^*]$ in V_{res}, which can be

achieved through a linear scan over V_{res}. In the example shown in Figure 4.3, the vectors $V^1 = V_3^1$ and $V^2 = V_5^2$ are used to compute the result vector

$$V_{\text{res}} = V^1 \wedge V^2 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \wedge \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{matrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{matrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}. \tag{4.6}$$

V_{res} has only one set bit at the third position corresponding to rule R_3, which is indeed the most highly prioritized matching rule, as the figure confirms.

The memory requirements and the preprocessing time required by BV are at most quadratic in the number of rules, since every dimension may yield $2n + 1$ bit vectors of n bits each. The classification operation requires d bit vector retrievals, which can generally be implemented as binary searches due to the ordering of the intervals. Furthermore, the computation of the final result vector as well as finding the first set bit require time linear in the number of rules. Still, BV often performs significantly faster than both Linear Search and Tuple Space Search, because many of its operations can be efficiently vectorized even on general purpose CPU systems [10]. If we denote the *machine word width* by w (which is typically 32 or 64 bit in off-the-shelf systems), then every vector can be represented by $\lceil \frac{n}{w} \rceil$ machine words, Therefore, a practical implementation can perform $d-1$ wordwise AND operations on words of the partial vectors in order to compute one word of the result vector V_{res}. Subsequently, a single comparison instruction can decide whether at least one bit in the result word is set. Only if this is the case, the w bits in the word must be checked, otherwise the next result word can be computed, which effectively allows to traverse large parts the result vectors wordwise rather than bitwise. In contrast to the previously discussed algorithms, BV does not support for quick incremental updates, because the addition or removal of rules requires an adjustment of every single bit vector and thus results in an effort quadratic in n. Hence, a change in the rule set typically requires a rebuild of the search data structure. Table 4.4 provides an overview over the key performance characteristics of BV.

Classification operation	Data structure creation	Data structure update	Memory requirements
$\mathcal{O}(d \cdot \log(n) + \lceil \frac{n}{w} \rceil)$	$\mathcal{O}(d \cdot n^2)$	$\mathcal{O}(d \cdot n^2)$	$\mathcal{O}(d \cdot n^2)$

n: number of rules d: number of fields w: machine word width

Tab. 4.4: Bit Vector Search performance characteristics.

Due to its decompositional nature, BV is suitable for implementation on a wide variety of hardware platforms, because the bit vector retrieval operations can be solved independently and therefore in parallel [68, 151]. Also, some platforms such as ASICs or FPGAs provide support for large machine word widths $w \gg 64$ and can compute the first set index in the result vector in logarithmic time, which further reduces the classification latency [56, 15], as explained in Section 15.2.

4.3.2 Aggregated Bit Vector Search

The original BV algorithm described in Section 4.3.1 may require $d \cdot \lceil \frac{n}{w} \rceil$ memory accesses to perform a single classification operation, because each word w_i^{res} of the final result vector is computed by ANDing the words $w_i^j, j \in \{, 1, \ldots, d\}$, from the one-dimensional vectors at the corresponding positions. However, the bit vectors that are generated from rule sets are often sparse, in the sense that large parts of the vectors do not contain any set bits [31]. This, in turn, can lead to a situation where at least one of the d words w_i^j that are used to compute w_i^{res} consists entirely of unset bits. As a consequence, the inspection of w_i^{res} will not terminate the classification process, as every bit in w_i^{res} will also be unset.

To illustrate this situation, Figure 4.4 shows two sparse example bit vectors V^1 and V^2 for a two-dimensional rule set with 16 rules. Assuming a machine word width $w = 2$, a total of 14 memory accesses is required to locate the first set bit in the result vector V^{res}, which is stored in the word w_7^{res}. Of these 14 memory accesses, the first 12 are used to compute result words that are entirely unset and only the last two accesses lead to a non-zero result word.

As noticed by Baboescu and Varghese [31], these situations can occur frequently with a growing number of rules and dimensions. They argue that one sparse dimension is sufficient to lead to situations such as the one described above, and

$$
\begin{array}{lcl}
V^1 & = & \big[\ \underbrace{0\ 0}_{w_1^1}\ \underbrace{0\ 0}_{w_2^1}\ \underbrace{1\ 0}_{w_3^1}\ \underbrace{0\ 0}_{w_4^1}\ \underbrace{0\ 0}_{w_5^1}\ \underbrace{0\ 0}_{w_6^1}\ \underbrace{1\ 0}_{w_7^1}\ \underbrace{0\ 0}_{w_8^1}\ \big] \\
V^2 & = & \big[\ \underbrace{0\ 0}_{w_1^2}\ \underbrace{0\ 0}_{w_2^2}\ \underbrace{0\ 1}_{w_3^2}\ \underbrace{0\ 0}_{w_4^2}\ \underbrace{0\ 0}_{w_5^2}\ \underbrace{0\ 0}_{w_6^2}\ \underbrace{1\ 1}_{w_7^2}\ \underbrace{0\ 1}_{w_8^2}\ \big] \\
\bigwedge & & \\
\hline
V^{\text{res}} & = & \big[\ \underbrace{0\ 0}_{w_1^{\text{res}}}\ \underbrace{0\ 0}_{w_2^{\text{res}}}\ \underbrace{0\ 0}_{w_3^{\text{res}}}\ \underbrace{0\ 0}_{w_4^{\text{res}}}\ \underbrace{0\ 0}_{w_5^{\text{res}}}\ \underbrace{0\ 0}_{w_6^{\text{res}}}\ \underbrace{1\ 0}_{w_7^{\text{res}}}\ \underbrace{0\ 0}_{w_8^{\text{res}}}\ \big]
\end{array}
$$

Fig. 4.4: Sparse bit vectors.

therefore propose the *Aggregated Bit Vector Search (ABV)*, an enhanced variant of the original BV scheme [31]. The main idea of the ABV scheme is to use *bit vector aggregation* to avoid a large number of memory accesses to empty words. The usage of aggregated bit vectors allows to traverse sparse parts of the original search data structure more quickly, at the cost of a slightly increased memory requirement for the search data structure. To this end, for every bit vector V in the original BV algorithm, the ABV approach computes an additional vector V_{agg} of length $\lceil \frac{n}{a} \rceil$ in the preprocessing phase, where a is a predefined *aggregation size*. For an aggregated vector V_{agg}, each bit $V_{\text{agg}}[i]$ is set iff at least one of the a corresponding bits $V[a \cdot i]$ to $V[a \cdot (i+1) - 1]$ is set, i. e.,

$$\forall\, i \in \left\{1, \ldots, \left\lceil \frac{n}{a} \right\rceil \right\} : V_{\text{agg}}[i] = \bigvee_{j=a \cdot i}^{\min\{a \cdot (i+1)-1, n\}} V[j]\,. \tag{4.7}$$

Figure 4.5 shows the aggregated vectors V^1_{agg} and V^2_{agg} for the original vectors V^1 and V^2 with an aggregation size of $a = 2$. It can be seen that for every word w^i_j from the original vectors V^i that is aggregated, there exists one bit b^i_j in the corresponding aggregated vector V^i_{agg}, which is an important property for the classification phase.

The classification phase of ABV is similar to the one of the Lucent scheme, with the main difference that for each dimension j, an aggregated vector V^j_{agg} is retrieved in addition to the original vector V^j. However, this time, the words u^j_i of the aggregated vectors V^j_{agg} are bitwise ANDed instead of the original vectors, as shown in Figure 4.5. As before, the goal is to find the first word in the aggregated result vector $V^{\text{res}}_{\text{agg}}$ with a set bit. The existence of such a bit b_k at position k implies that in each original vector V^j, the kth aggregated word w^j_k contains at least one set bit. Therefore, as in the Lucent scheme, the word w^{res}_k is computed by bitwise ANDing the words w^j_k and subsequently inspected for a set bit. If such a

$$\begin{array}{lcl}
 & & \begin{array}{cccccccc} w^1_1 & w^1_2 & w^1_3 & w^1_4 & w^1_5 & w^1_6 & w^1_7 & w^1_8 \end{array} \\
V^1 & = & \left[\begin{array}{cccccccc} \underbrace{0\ 0} & \underbrace{0\ 0} & \underbrace{1\ 0} & \underbrace{0\ 0} & \underbrace{0\ 0} & \underbrace{0\ 0} & \underbrace{1\ 0} & \underbrace{0\ 0} \end{array} \right] \\
V^1_{\text{agg}} & = & \left[\begin{array}{cccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ b^1_1 & b^1_2 & b^1_3 & b^1_4 & b^1_5 & b^1_6 & b^1_7 & b^1_8 \end{array} \right] \\
 & & \begin{array}{cccccccc} w^2_1 & w^2_2 & w^2_3 & w^2_4 & w^2_5 & w^2_6 & w^2_7 & w^2_8 \end{array} \\
V^2 & = & \left[\begin{array}{cccccccc} \underbrace{0\ 0} & \underbrace{0\ 0} & \underbrace{0\ 1} & \underbrace{0\ 0} & \underbrace{0\ 0} & \underbrace{0\ 0} & \underbrace{1\ 1} & \underbrace{0\ 1} \end{array} \right] \\
V^2_{\text{agg}} & = & \left[\begin{array}{cccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ b^2_1 & b^2_2 & b^2_3 & b^2_4 & b^2_5 & b^2_6 & b^2_7 & b^2_8 \end{array} \right]
\end{array}$$

Fig. 4.5: Computing aggregated bit vectors.

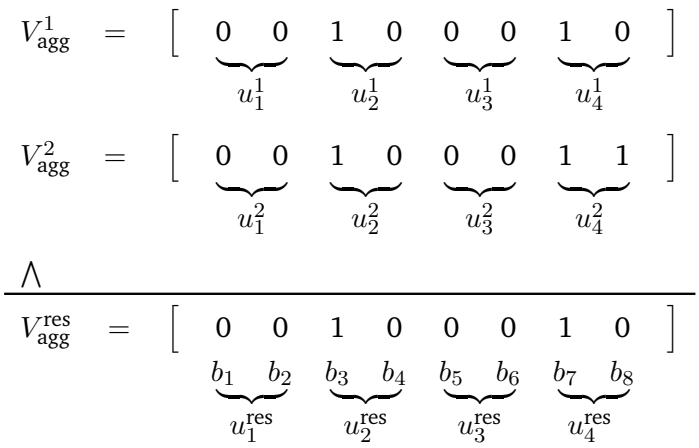


Fig. 4.6: ANDing aggregated bit vectors.

bit exists, the classification terminates, otherwise, the search in the aggregated vector continues. In the example in Figure 4.6, it takes a total of 12 memory accesses until the search terminates, it contrast to the 14 accesses of the Lucent scheme, since all eight words u^j_i as well as w^1_3, w^2_3, w^1_7 and w^2_7 are accessed. Note that a set bit in the aggregated result vector can lead to a false positive lookup of the corresponding words in the original vectors. For example, both words w^1_3 and w^2_3 in Figure 4.5 result in the aggregation bits $b^1_3 = b^2_3 = 1$. This, in turn, leads to the inspection of w^1_3 and w^2_3, which do not have any common set bits.

The worst case performance of the ABV classification algorithm is not different from the BV worst case performance. Nevertheless, in practice, ABV can outperform BV significantly due to the ability to quickly traverse gaps of unset bits. Furthermore, the authors of [31] suggest an additional enhancement to their ABV technique, which uses a rule sorting mechanism to group rules in a way that reduces the likelihood for false positive lookups. However, this requires the algorithm to compute all matching rules and not only the first one, as rule re-ordering violates rule prioritization and therefore typically the rule set's semantics.

4.4 Crossproducting Approaches

Similar to bit vector search algorithms presented in Section 4.3, crossproducting approaches are decompositional classification algorithms that perform independent lookups on the distinct header values of an incoming packet. The independent lookup results are then combined in order to obtain the final classification result.

Nr. / Priority	Field F_1	Field F_2	Action
R_1	$[2, 5]$	$[3, 3]$	a^1
R_2	$[4, 4]$	$[2, 4]$	a^2
R_3	$[4, 4]$	$[6, 6]$	a^3

Tab. 4.5: A two-dimensional example rule set over $\mathcal{H} = [0, 7]^2$.

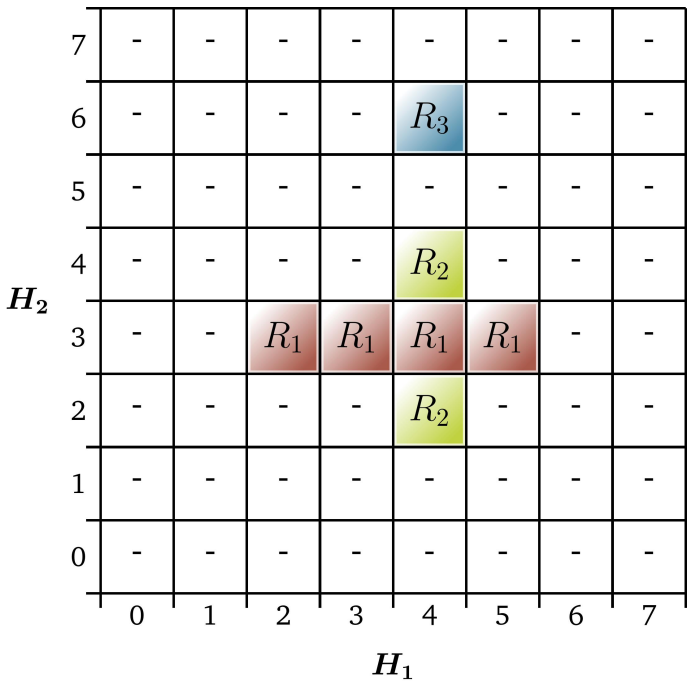


Fig. 4.7: A complete crossproduct table for the rule set given in Table 4.5.

The motivation behind crossproducting schemes is the fact that the fastest possible way to conduct a packet classification is to perform a single direct table lookup using the packet's relevant header fields. If we could construct a d-dimensional table $T_{\mathfrak{R}}$ for a rule set $\mathfrak{R}$ such that every possible header combination $(h_1, \ldots, h_d)$ in the header space $H_1 \times \ldots \times H_d$ is a valid key in $T_{\mathfrak{R}}$ with $T_{\mathfrak{R}}[(h_1, \ldots, h_d)] = R_{i^*}$, then every classification operation would only require exactly one memory access. An example crossproduct table for the rule set defined in Table 4.5 is shown in Figure 4.7. Unfortunately, such a table requires $\prod_{i=1}^{d} |H_i|$ entries, which is infeasible in most cases. For example, already a two-dimensional table for source and destination IPv4 addresses needs to provide 2^{64} entries and thus cannot be implemented on current systems. Therefore, practical crossproducting schemes, such as *RFC* [62] and *ODC* [128], do not precompute a single large lookup table, but instead rely on multiple tables and caching, as we will see in Sections 4.4.1 and 4.4.2, respectively.

4.4.1 Recursive Flow Classification

The *Recursive Flow Classification (RFC)* approach [62] by Gupta and McKeown is a decompositional classification algorithm that distinguishes itself from the previously discussed bit vector approaches in the following key points: first, RFC does not perform binary searches, but exclusively relies on direct table lookups. Second, RFC essentially moves the search for the first set bit, as it is performed by bit vector searches, to the preprocessing phase, in which the RFC data structure is constructed. Therefore, classification operations can be executed significantly faster, once the RFC structure is available. However, RFC's boost in classification performance comes at the cost of tremendously long preprocessing times as well as huge memory requirements even for moderately sized rule sets [10].

In order to prevent the memory footprint of a complete crossproduct table while still being able to classify a packet with a fixed number of operations, RFC splits the lookup process into multiple *phases* $P_1, \ldots, P_\psi$. We start to detail the RFC algorithm by first giving a general overview over the phased classification process. Subsequently, we provide a detailed two-dimensional example based on the rule set given in Table 4.5.

The number of RFC phases ψ depends on the number of relevant header fields d as well as on the predefined *table join factor* constant J and is computed by $\psi = \lceil \log_J(d) \rceil$. In the jth phase P_j, $\beta_i = \lfloor \frac{d}{2^{(j-1)}} \rfloor$ lookup tables T_i^j are accessed via lookup keys κ_i^j. Here, each table T_i^j stores a multiset of so-called *equivalence IDs*. Every lookup operation $T_i^j \left[\kappa_i^j\right]$ yields an *equivalence ID* $\mathcal{E}_i^j$ that refers to a corresponding set of rules which could match the scrutinized packet. In the first $\psi - 1$ stages, these equivalence IDs are used to compute the lookup keys $\kappa_i^{(j+1)} (i \in \{1, \ldots, \beta_{j+1}\})$ that are used in the next phase P_{j+1}. However, the equivalence ID $\mathcal{E}_1^\psi$ that is obtained in the final stage P_ψ is the index of the most highly prioritized matching rule R_{i^*}, i. e., $i^* = \mathcal{E}_1^\psi$. The RFC classification process as well as the lookup key computation is sketched in Figure 4.8 for a four-dimensional rule set and a table join factor of two. In the first phase P_1, each lookup key κ_i^1 is set to the corresponding header value h_i^p of the currently classified packet p. It is important to note that the RFC classification requires a constant number of operations for a fixed number of dimensions d, regardless of the rule set's size. Figure 4.8 confirms this fact for $d = 4$: any incoming packet can be classified using seven memory accesses (four in phase P_1, two in phase P_2, one in phase P_3).

In order to initially construct the lookup tables T_i^1, the RFC preprocessing step starts off the same way as the bit vector preprocessing, as described in Section 4.3.

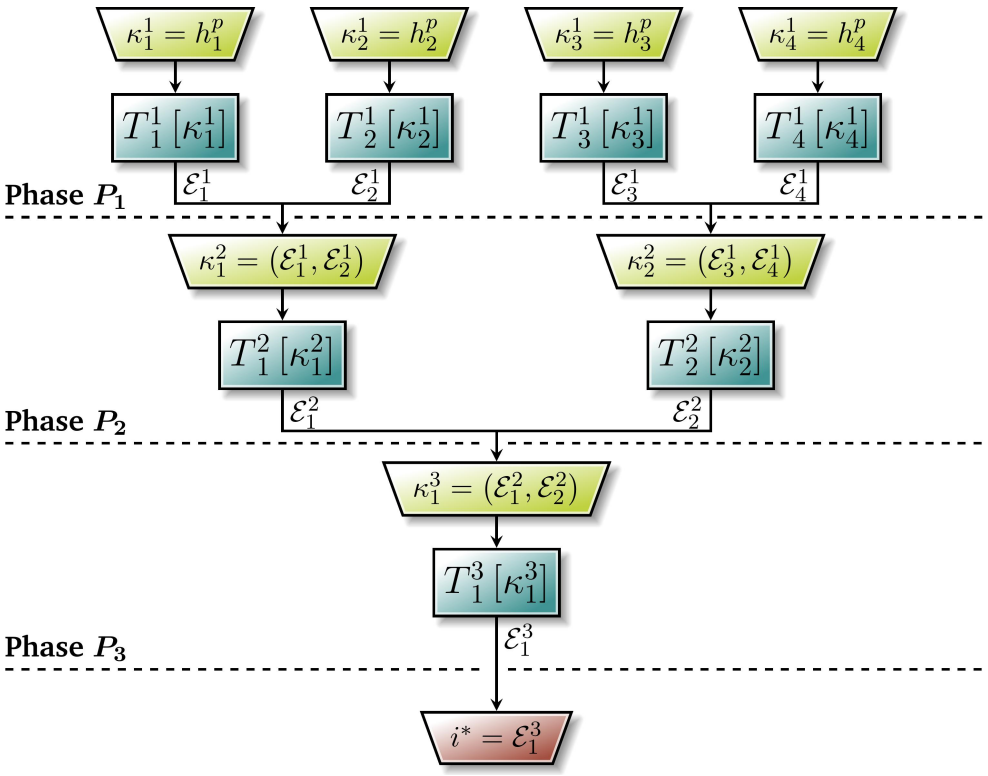


Fig. 4.8: Example for the RFC classification flow $(d = 4, J = 2)$.

That is, for all $j \in \{1, \ldots, d\}$, the jth axis of the header space bounding box is partitioned into α_j intervals I_k^j with associated bit vectors V_k^j. Next, an equivalence ID $\mathcal{E}_{k,j}^1$ is assigned to every bit vector, with the restriction that

$$\mathcal{E}_{k,j}^1 = \mathcal{E}_{k',j}^1 \Leftrightarrow V_k^j = V_{k'}^j. \tag{4.8}$$

The reasoning behind this assignment is that packets, which fall into different intervals but are mapped onto the same set of possibly matching rules (as indicated by the corresponding vectors), belong to the same equivalence group. Because the equivalence IDs are used as indices for array lookups during packet classification, their numbering begins at zero and is increased by one for every new distinct bit vector. Finally, for each dimension j, every table T_j^1 is constructed by allocating an array of size $|H_j|$, such that for every interval $I_k^j = [a, b]$ the following condition holds:

$$\forall\, \kappa \in [a, b] : T_j^1[\kappa] = \mathcal{E}_{k,j}^1. \tag{4.9}$$

Note that every interval corresponds to exactly one equivalence ID and therefore, the above condition is unambiguous. Figures 4.9 and 4.11 illustrate the computation of the lookup tables T_1^1 and T_2^1 from the rule set shown in Table 4.5. Furthermore, we keep track of the set of unique bit vectors $\mathcal{V}_j^1$ that was used for the construction of each table T_j^1. In our example, $\mathcal{V}_1^1 = \{V_1^1, V_2^1, V_3^1\}$ and $\mathcal{V}_2^1 = \{V_1^2, V_2^2, V_3^2, V_6^2\}$. Of course, the construction of an initial table T_j^1 is only

Vectors:	$\boldsymbol{V_1^1}$ $\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\boldsymbol{V_2^1}$ $\begin{bmatrix}1\\0\\0\end{bmatrix}$	$\boldsymbol{V_3^1}$ $\begin{bmatrix}1\\1\\1\end{bmatrix}$	$\boldsymbol{V_4^1}$ $\begin{bmatrix}1\\0\\0\end{bmatrix}$	$\boldsymbol{V_5^1}$ $\begin{bmatrix}0\\0\\0\end{bmatrix}$
Intervals:	$\boldsymbol{I_1^1}$ $[0,1]$	$\boldsymbol{I_2^1}$ $[2,3]$	$\boldsymbol{I_3^1}$ $[4,4]$	$\boldsymbol{I_4^1}$ $[5,5]$	$\boldsymbol{I_5^1}$ $[6,7]$
Eq. IDs:	$\boldsymbol{\mathcal{E}_{1,1}^1}$ 0	$\boldsymbol{\mathcal{E}_{2,1}^1}$ 1	$\boldsymbol{\mathcal{E}_{3,1}^1}$ 2	$\boldsymbol{\mathcal{E}_{4,1}^1}$ 1	$\boldsymbol{\mathcal{E}_{5,1}^1}$ 0

(a) Vectors, intervals, and eq. IDs.

$T_1^1 =$

κ	Eq. ID
0	0
1	0
2	1
3	1
4	2
5	1
6	0
7	0

(b) The lookup table T_1^1.

Fig. 4.9: Computing the lookup table T_1^1 for field F_1 for the rule set in Table 4.5.

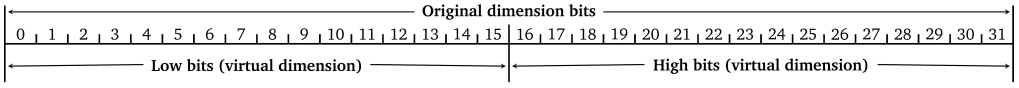


Fig. 4.10: Splitting a large dimension into smaller sub-dimensions..

practical when the number of possible header values $|H_j|$ is not too large. For example, the table for an IPv4 header field requires 2^{32} entries, which is infeasible on many systems. Therefore, such large dimensions are split into virtual sub-dimensions such that each sub-dimension can be mapped to an initial table, as sketched in Figure 4.10.

In the subsequent phases $P_2, \ldots, P_\psi$, the lookup tables for the corresponding phase are constructed by combining the bit vectors from direct predecessor stages. For a given phase P_j with β_j lookup tables, the contents of the ith table T_i^j depends on the sets of unique bit vectors $\mathcal{V}_{(i-1)\cdot J+1}^{j-1}, \ldots, \mathcal{V}_{i\cdot J}^{j-1}$ that were created during the generation of the tables $T_{(i-1)\cdot J+1}^{j-1}, \ldots, T_{i\cdot J}^{j-1}$. More precisely, for each combination $(V_{l_1}, \ldots, V_{l_J}) \in \mathcal{V}_{(i-1)\cdot J+1}^{j-1} \times \ldots \times \mathcal{V}_{i\cdot J}^{j-1}$, a new bit vector $V_{l_1,\ldots,l_J}$ is computed through bitwise ANDing, i. e.,

$$V_{l_1,\ldots,l_J} = V_{l_1} \wedge \ldots \wedge V_{l_J}. \tag{4.10}$$

As in the first phase P_1, an equivalence ID $\mathcal{E}_{l_1,\ldots,l_J,i}^j$ is assigned to the newly computed vector $V_{l_1,\ldots,l_J}$. In the case that $j = \psi$, i. e., the table for the last stage is generated, then $\mathcal{E}_{l_1,\ldots,l_J,i}^j$ is equal to the index of the first set bit in $V_{l_1,\ldots,l_J} = V_{l_1}$. Otherwise, if the vector $V_{l_1,\ldots,l_J}$ is equal to a previously computed vector $V_{l'_1,\ldots,l'_J}$, then $\mathcal{E}_{l_1,\ldots,l_J,i}^j$ is set to $\mathcal{E}_{l'_1,\ldots,l'_J,i}^j$, else $\mathcal{E}_{l_1,\ldots,l_J,i}^2$ is assigned a new equivalence ID, just as in phase P_1. Finally, $T_i^j\,[(l_1, \ldots, l_J)]$ is set to $\mathcal{E}_{l_1,\ldots,l_J,i}^j$. Figure 4.12 sketches the computation of the ANDed bit vectors as well as the equivalence ID assignment. The resulting lookup table T_1^2 is shown in Figure 4.13. It can be seen that the memory footprint of all three lookup tables generated by RFC, which require 8 + 8 + 12 = 28 entries in total, are smaller then the single crossproduct table given in Figure 4.7 with 64 entries. Still, the RFC tables can be used to

Vectors:	$\boldsymbol{V_1^2}$ $\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\boldsymbol{V_2^2}$ $\begin{bmatrix}0\\1\\0\end{bmatrix}$	$\boldsymbol{V_3^2}$ $\begin{bmatrix}1\\0\\0\end{bmatrix}$	$\boldsymbol{V_4^2}$ $\begin{bmatrix}0\\1\\0\end{bmatrix}$	$\boldsymbol{V_5^2}$ $\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\boldsymbol{V_6^2}$ $\begin{bmatrix}0\\0\\1\end{bmatrix}$	$\boldsymbol{V_7^2}$ $\begin{bmatrix}0\\0\\0\end{bmatrix}$
Intervals:	$\boldsymbol{I_1^2}$ $[0,1]$	$\boldsymbol{I_2^2}$ $[2,2]$	$\boldsymbol{I_3^2}$ $[3,3]$	$\boldsymbol{I_4^2}$ $[4,4]$	$\boldsymbol{I_5^2}$ $[5,5]$	$\boldsymbol{I_6^2}$ $[6,6]$	$\boldsymbol{I_7^2}$ $[7,7]$
Eq. IDs:	$\boldsymbol{\mathcal{E}_{1,2}^1}$ 0	$\boldsymbol{\mathcal{E}_{2,2}^1}$ 1	$\boldsymbol{\mathcal{E}_{3,2}^1}$ 2	$\boldsymbol{\mathcal{E}_{4,2}^1}$ 1	$\boldsymbol{\mathcal{E}_{5,2}^1}$ 0	$\boldsymbol{\mathcal{E}_{6,2}}$ 3	$\boldsymbol{\mathcal{E}_{7,2}^1}$ 0

(a) Vectors, intervals, and eq. IDs.

$T_2^1 =$

κ	Eq. ID
0	0
1	0
2	1
3	2
4	1
5	0
6	3
7	0

(b) The lookup table T_2^1.

Fig. 4.11: Computing the lookup table T_2^1 for field F_2 for the rule set in Table 4.5.

$V_1^1 \wedge V_6^2 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \mathcal{E}_1^2 = 0$	$V_2^1 \wedge V_6^2 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \mathcal{E}_1^2 = 0$	$V_3^1 \wedge V_6^2 = \begin{bmatrix}0\\0\\1\end{bmatrix}, \mathcal{E}_1^2 = 3$
$V_1^1 \wedge V_3^2 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \mathcal{E}_1^2 = 0$	$V_2^1 \wedge V_3^2 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \mathcal{E}_1^2 = 1$	$V_3^1 \wedge V_3^2 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \mathcal{E}_1^2 = 1$
$V_1^1 \wedge V_2^2 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \mathcal{E}_1^2 = 0$	$V_2^1 \wedge V_2^2 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \mathcal{E}_1^2 = 0$	$V_3^1 \wedge V_2^2 = \begin{bmatrix}0\\1\\0\end{bmatrix}, \mathcal{E}_1^2 = 2$
$V_1^1 \wedge V_1^2 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \mathcal{E}_1^2 = 0$	$V_2^1 \wedge V_1^2 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \mathcal{E}_1^2 = 0$	$V_3^1 \wedge V_1^2 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \mathcal{E}_1^2 = 0$

Fig. 4.12: Computing the eq. IDs for lookup table T_1^2 for the rule set from Table 4.5.

classify any packet (h_1, h_2) within $\mathcal{H}$ with three memory accesses. For example, the header $(4, 3)$ would be classified by first looking up the equivalence IDs $\mathcal{E}_1^1 = T_1^1[4] = 2$ and $\mathcal{E}_2^1 = T_2^1[3] = 2$. Subsequently, the key $\kappa_1^2 = (2, 2)$ is used to lookup $\mathcal{E}_1^2 = T_1^2[(2, 2)] = 1$, which is the index of the most highly prioritized matching rule.

Clearly, the storage requirements of RFC are the sum of all utilized tables' sizes, i. e.,

$$\sum_{j=1}^{\lceil \log_J(d) \rceil} \sum_{i=1}^{\beta_j} |T_i^j|. \tag{4.11}$$

The sizes of the tables T_i^1 in phase P_1 are $|H_i|$, respectively, and each of these tables contains at most $2n + 1$ distinct equivalence IDs due to the point projection

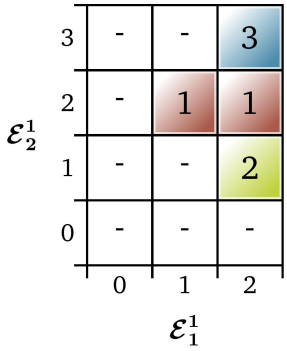


Fig. 4.13: The table T_1^2 for the rule set from Table 4.5 is accessed with $\kappa_1^2 = (\mathcal{E}_1^1, \mathcal{E}_2^1)$.

procedure in each dimension. The size of every table T_i^j in a later phase (i. e., $j > 1$) is the product of the number of distinct equivalence IDs of its predecessor tables, and every table is a predecessor to at most one other table. Therefore, the number entries in the final table T_1^ψ is in $\mathcal{O}(n^d)$, as is Sum 4.11. Because these tables must be allocated and written entry by entry in the preprocessing step, the entire data structure construction takes $\mathcal{O}(n^d)$ time as well. Unfortunately, RFC does not support incremental data structure updates, as a rule insertion or deletion typically leads to a change in the phase one tables, which must be propagated to all successor tables. Finally, an RFC classification process requires one lookup in every table and thus requires at most

$$\begin{aligned}\sum_{i=1}^{\lceil \log_J(d) \rceil} \frac{d}{J^{i-1}} &= d \cdot \sum_{i=1}^{\lceil \log_J(d) \rceil} \frac{1}{J^{i-1}} \\ &\leq d \cdot \sum_{i=0}^{\lceil \log_J(d) \rceil} \frac{1}{J^i} \\ &\leq d \cdot \sum_{i=0}^{\infty} \frac{1}{J^i} \\ &= d \cdot \frac{J}{J-1}\end{aligned} \tag{4.12}$$

memory accesses, plus a few arithmetic operations in the same order of magnitude. Given that J is a small predefined constant, it follows that RFC's classification operation is in $\mathcal{O}(d)$. Furthermore, it should be noted that the RFC classification process can be implemented without branches for a fixed d. These performance characteristics are summarized in Table 4.6.

Generally, RFC is considered to be one of the fastest existing classification algorithms [61, 163]. However, its high storage requirements and long preprocessing times diminish its applicability for environments with large rule sets or frequently changing rule sets [10]. The *Hierarchical Space Mapping (HSM)* technique [147] works similar to RFC, but replaces the table lookups in RFC's first stage with

Classification operation	Data structure creation	Data structure update	Memory requirements
$\mathcal{O}(d)$	$\mathcal{O}(n^d)$	$\mathcal{O}(n^d)$	$\mathcal{O}(n^d)$

n: number of rules d: number of fields

Tab. 4.6: Recursive Flow Classification performance characteristics.

binary searches, which mitigates the storage requirements to some extent. An enhanced version of RFC, ERCF [58], aims to tackle this problem by using the bit vector aggregation techniques from the ABV [31] in order to accelerate the preprocessing phase. The main idea here is that the bit vector intersections, that are required in the RFC table generation, can be executed quicker when using aggregated bit vectors.

4.4.2 On Demand Crossproducting

A more dynamic crossproducting scheme than RFC was proposed Srinivasan, Varghese, Suri, and Waldvogel under the name *On Demand Crossproducting (ODC)* [128]. In contrast to RFC, the ODC scheme relies on two rather than ψ phases and the lookups in the first phase are executed on trie data structures [51] rather than lookup tables. The most striking difference to RFC, however, is the on demand construction of the lookup data structure T_1^2 used in the second phase: instead of precomputing the entire structure T_1^2 before the classification process, ODC incrementally adds entries to T_1^2 during packet classification. Thus, T_1^2 is essentially used as a cache for the tuple of equivalence IDs $(\mathcal{E}_1^1, \ldots, \mathcal{E}_d^1)$ that results from the individual one-dimensional lookups in the first stage. Accordingly, the table T_1^2 is required to be updateable quickly, and is therefore implemented as a hash table.

ODC's preprocessing step is considerably simpler than RFC's, since ODC only requires the construction of d one-dimensional tries as well as an empty hash map. Hence, the preprocessing time is in $\mathcal{O}\left(n \cdot \sum_{j=1}^{d} w_j\right)$, where w_j is the minimum number of required bits to represent an arbitrary value in H_j. Of course, in principle any other adequate longest prefix matching technique can be used for the one-dimensional structures, for example multibit tries [60, 116] or hash-based schemes [141]. If a w_j is sufficiently small, e. g., $w_j \leq 16$, then the corresponding trie T_j^1 can be implemented as a direct lookup table, which technically also is a multibit trie. However, for the remainder of this section, we assume that every T_j^1 is implemented as a unibit trie.

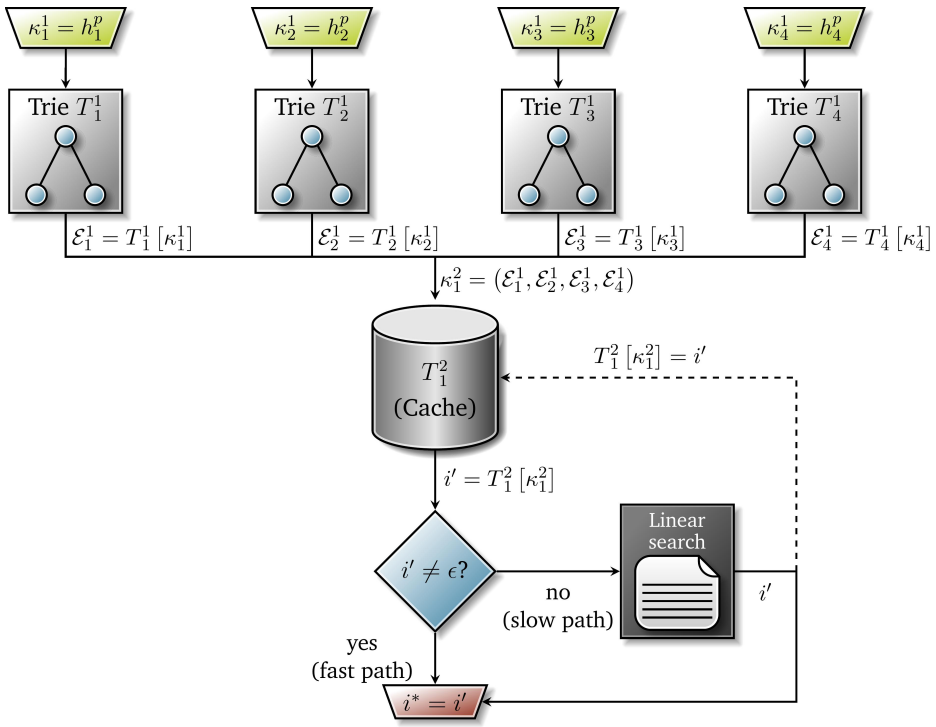


Fig. 4.14: Sketch of a four-dimensional ODC classification process.

In order to classify an incoming packet p with the header $(h_1^p, \ldots, h_d^p)$, each trie T_j^1 is queried with the corresponding header value h_j^p for an equivalence ID $\mathcal{E}_j^1$, which requires at most $\mathcal{O}\left(\sum_{j=1}^{d} w_j\right)$ memory lookups. Subsequently, the algorithm aims to determine the index i^* of the most highly prioritized matching rule R_{i^*} with a hash lookup in the cache T_1^2, i. e.,

$$i^* = T_1^2\left[\mathcal{E}_1^1, \ldots, \mathcal{E}_d^1\right]. \tag{4.13}$$

In the case of a cache miss, i. e., $i^* = \epsilon$, ODC performs a linear search in the rule set $\mathfrak{R}$ to determine i^* and updates the cache with

$$T_1^2\left[\mathcal{E}_1^1, \ldots, \mathcal{E}_d^1\right] = i^*. \tag{4.14}$$

Clearly, this strategy requires amortized $\mathcal{O}(1)$ time in case of a cache hit, and otherwise $\mathcal{O}(d \cdot n)$ time due to the linear search. The ODC classification operation is depicted in Figure 4.14.

While ODC does not reach RFC's excellent classification performance, it provides one key advantage over RFC: namely its capability of incremental updates to the search data structure. If a rule is added to or deleted from the rule set $\mathfrak{R}$ implemented by ODC, the one-dimensional tries T_j^1 can be locally adjusted with $\mathcal{O}(w_j)$ operations per trie. Unfortunately, an incremental update can invalidate existing cache entries, which at least requires a partial cache reset. However,

Classification operation	Data structure creation	Data structure update	Memory requirements
$\mathcal{O}(n)$	$\mathcal{O}\left(n \cdot \sum_{j=1}^{d} w_j\right)$	$\mathcal{O}\left(\sum_{j=1}^{d} w_j + \tau\right)$	$\mathcal{O}\left(\sum_{j=1}^{d} w_j + \tau\right)$

n: number of rules d: number of fields τ: cache threshold

Tab. 4.7: On Demand Crossproducting performance characteristics.

the cost of this operation can be diminished when the cache size is bounded by at constant threshold τ. Of course, this may have a negative impact on the classification performance in case of thrashing occurrences. The ODC performance characteristics are listed in Table 4.7.

4.5 Decision Tree Algorithms

The last class of classification algorithms we dive into in this chapter are *decision tree* approaches, which are, much like the bit vector and RFC approaches we saw earlier, based on the geometric representation of the rule set. As the name suggests, these algorithms translate the specified rule set into one or several multi-dimensional decision trees in a preprocessing step, that are subsequently used to classify incoming network packets. The decision tree creation is guided by several heuristics, which differ between the various different existing approaches [55, 63, 83, 90, 107, 115, 122, 137, 146]. The basic idea is to recursively *cut* the header space $\mathcal{H}$, which contains the geometric representations of all rules from the rule set $\mathfrak{R}$, into smaller hyperrectangles B_l with $\bigcup B_l = \mathcal{H}$ such that every B_l hopefully intersects with a smaller number of rules. Each area created by a cut operation represents a node in the decision tree that is eventually used as the search data structure. The recursion terminates in a node when the number of rules that intersect with that node is smaller than a predefined threshold. This process is sketched in Figure 4.15.

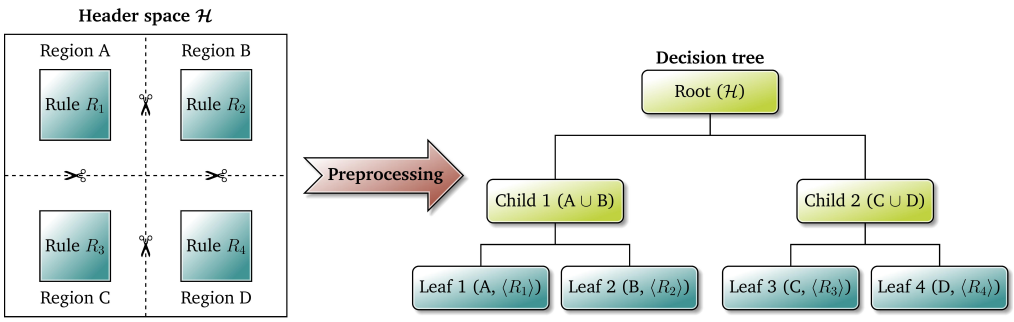


Fig. 4.15: The principle of decision tree creation.

Each node in the generated decision tree contains information about the region it covers. If a node is a non-leaf node, it stores where the covered region has been cut, which is essential in order to traverse the tree by using an incoming packet's header data. Finally, if a node is a leaf node, it contains a pointer to the sub rule set that is associated to the covered region. This sub rule set is searched linearly, once the traversal has reached a leaf node in the tree. Of course, the performance of decision tree approaches is dependent on the height of the generated tree(s) as well as the size of the sub rule sets located in the leaf nodes. In the following, we will depict two different decision tree algorithms: namely the seminal *HiCuts* [63] approach, which layed the foundation into this area of research, and the *HyperSplit* scheme [107], which significantly improves the layout of the generated search tree.

4.5.1 Hierarchical Intelligent Cuttings

Hierarchical Intelligent Cuttings (HiCuts) by Gupta and McKeown [63] recursively partitions the header space $\mathcal{H}$ through equidistant cuts, thereby creating a HiCuts decision tree. The root of the tree is associated with the hyperrectangle of the entire header space $B(\mathcal{H})$ and represents the vantage point of the recursive decision tree generation process. During preprocessing, a node N is cut if the hyperrectangle $B(N)$ that is associated to N covers more than β rules. Here, β is a predefined constant and is also called the *binth* [63] of the decision tree. Often, lower values of β result in deeper decision trees and faster searches of the child nodes during classification, whose sub rule sets are queried linearly. In contrast, high values of β lead to small decision trees that can be created quickly, which, however, can result in worse classification performance, as the linear searches become more expensive.

When a node N is cut, HiCuts uses various heuristics in order to decide (1) *which* dimension δ should be cut, and (2), *how many* cuts γ should be performed in dimension δ. Finding adequate values for both the cut dimension and the number of cuts is crucial for the performance of HiCuts, both in the preprocessing and in the classification phase, due to the fact that choosing bad values for, e. g., the cut dimension, can lead to memory explosions, huge preprocessing times, and bad classification performance. The original HiCuts paper [63] proposes four different heuristics that can be used to compute the cut dimension δ, but does not give a recommendation which of these heuristics should be used in which situations. We therefore describe only one of these heuristics, which was the most successful in our experiments and is also recommended for use in related work [122].

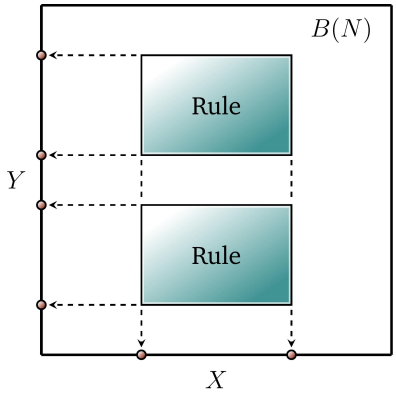


Fig. 4.16: The HiCuts dimension heuristic chooses $\delta = Y$ since $|\Pi_Y| = 4 > |\Pi_X| = 2$.

The idea of the *dimension choice heuristic* is to pick the dimension δ with the most *rule projection points*. That is, given a node N that covers the area $B(N) = \left[a_1^N, b_1^N\right] \times \ldots \times \left[a_d^N, b_d^N\right]$, all rules $R_i \in \mathfrak{R}$ are taken into consideration whose geometrical representation $B(R_i)$ intersects with $B(N)$. We refer to the sub rule set of the rules associated to N through intersection by $\mathfrak{R}_N$. Then, for each dimension j, the set of distinct rule projection points Π_j is computed with

$$\Pi_j = \left\{ x \;\middle|\; \begin{array}{l} \left(x = a_j^R \wedge x \in \left[a_j^N, b_j^N\right]\right) \vee \\ \left(x = b_j^R \wedge x \in \left[a_j^N, b_j^N\right]\right) \text{ with} \\ R \in \mathfrak{R}_N \text{ and } B(R)_j = \left[a_j^R, b_j^R\right] \end{array} \right\}. \tag{4.15}$$

Finally, the cutting dimension δ is the dimension with $|\Pi_\delta| \geq |\Pi_j| \; \forall j \in \{1, \ldots, d\}$. In case of a tie, the cutting dimension can be determined randomly from the best candidates. The dimension selection heuristic is illustrated in Figure 4.16.

After the cutting dimension δ has been determined for a node N, a *cut heuristic* chooses the number of cuts to perform in the dimension δ with respect to the bounds of N. Here, the HiCuts algorithm relies on a greedy strategy that starts off with $\max\{4, \lfloor\sqrt{|\mathfrak{R}_N|}\rfloor\}$ cuts, and then stepwise doubles the amount of cuts γ until the space required by the resulting child nodes exceeds a threshold Σ_N. More precisely, with C_N being the set of child nodes of N and *spfac* being a predefined *space factor*, the termination criterion for the greedy search is

$$\left(|C_N| + \sum_{C \in C_N} |\mathfrak{R}_C|\right) > (\mathit{spfac} \cdot |\mathfrak{R}_N|)\,. \tag{4.16}$$

Of course, γ is naturally limited by the amount of points in the interval $\left[a_\delta^N, b_\delta^N\right]$ of the area $B(N)$. Similar to the *binth* parameter, the space factor *spfac* influences the shape of the generated decision tree: the larger *spfac*, the more cuts can be performed, which can result in a shallower and broader tree. However, large

values for *spfac* can quickly lead to memory explosions, which is why the literature refers to small values of *spfac* of up to eight [63, 122].

Once the cut dimension δ and the number of cuts γ have been determined, the node N is partitioned into $\gamma + 1$ child nodes $C_1, \ldots, C_{\gamma+1}$. This happens by splitting the area

$$B(N) = \left[a_1^N, b_1^N\right] \times \ldots \times \left[a_\delta^N, b_\delta^N\right] \times \ldots \times \left[a_d^N, b_d^N\right] \tag{4.17}$$

into $\gamma + 1$ areas

$$B(C_i) = \left[a_1^N, b_1^N\right] \times \ldots \times \left[a_\delta^{C_i}, b_\delta^{C_i}\right] \times \ldots \times \left[a_d^N, b_d^N\right] \tag{4.18}$$

with $i \in \{1, \ldots, \gamma + 1\}$, $\bigcap_{i=1}^{\gamma+1} \left[a_\delta^{C_i}, b_\delta^{C_i}\right] = \emptyset$ and $\bigcup_{i=1}^{\gamma+1} \left[a_\delta^{C_i}, b_\delta^{C_i}\right] = \left[a_\delta^N, b_\delta^N\right]$. These areas are equally sized, except occasionally the last area $B(C_{\gamma+1})$.

Figure 4.17 depicts the above described HiCuts cutting process and thereby illustrates the major cause of memory usage in HiCuts, namely *rule duplication*. For example, the rule R_1 intersects with the areas of the children $C_1, \ldots C_4$ and is thus duplicated three times, as it must be processed in the child nodes. By looking at the figure, we could also assume that rule R_2 is duplicated into the children C_3, C_4, and C_5. However, in the case of child C_3, an important subtlety must be respected: note that within the area $B(C_3)$, the rule R_2 is completely covered by the rule R_1 and therefore can never be reached when the classification process enter C_3. Rule R_2 should therefore be removed from the child C_3, and generally,

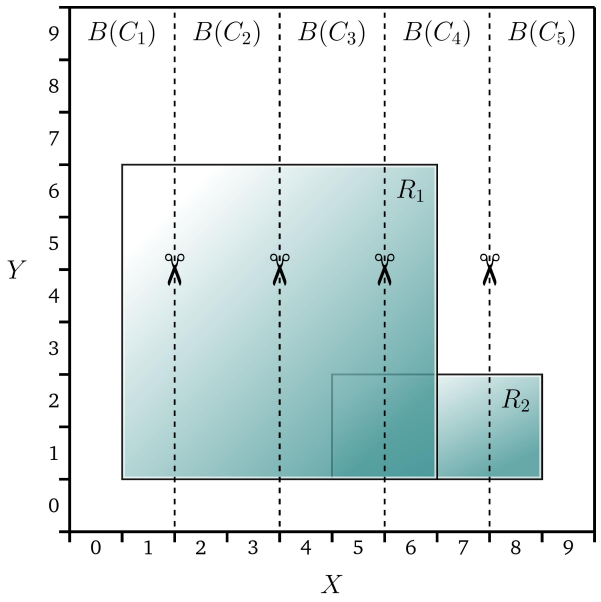


Fig. 4.17: The HiCuts cutting process creates five child nodes $C_1, \ldots, C_5$ by cutting four times in dimension X.

every rule in a node N that is completely covered by more highly prioritized rules should be removed from N. In the original HiCuts publication [63], this redundancy removal is mentioned as an optional and "time consuming" way to optimize the storage requirements of the resulting tree. However, in order to guarantee that the tree building process always terminates, this redundancy removal step is mandatory. As a simple example, consider a rule set $\mathfrak{R} = \langle R_1, R_2 \rangle$ with $B(R_1) = B(R_2)$. Without redundancy removal, it is not possible to construct a HiCuts tree for $\mathfrak{R}$ with a binth value $\beta = 1$, because no executed cut can separate R_1 from R_2.

Figures 4.18 and 4.19 illustrate a complete example for the HiCuts tree generation process. It can be seen that even for the small rule set in Figure 4.18 with five

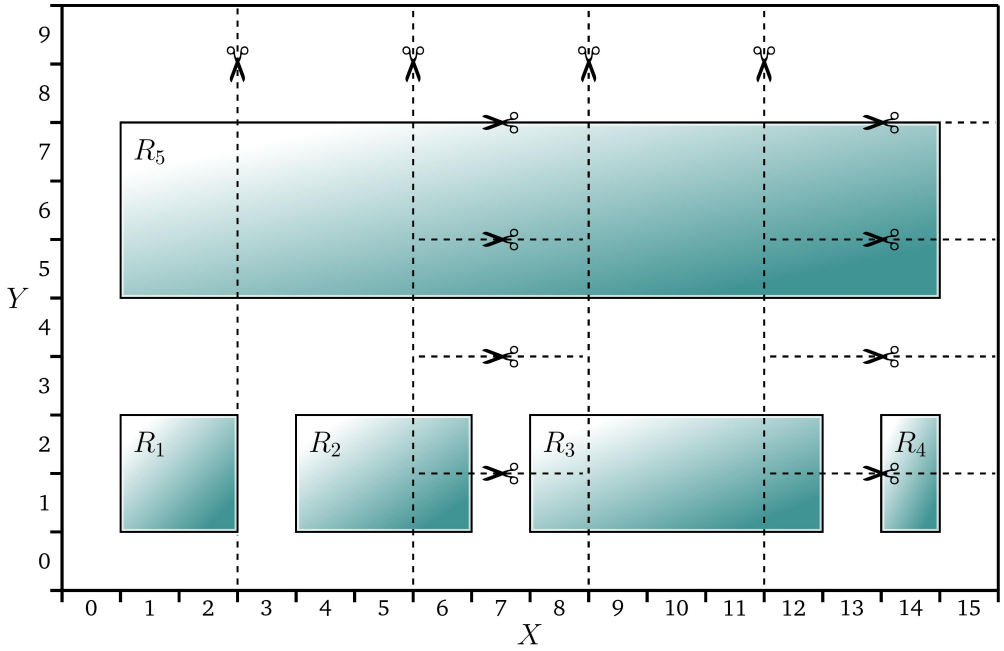


Fig. 4.18: Complete HiCuts example with five rules, $\beta = 2$, and $\mathcal{H} = [0, 15] \times [0, 9]$.

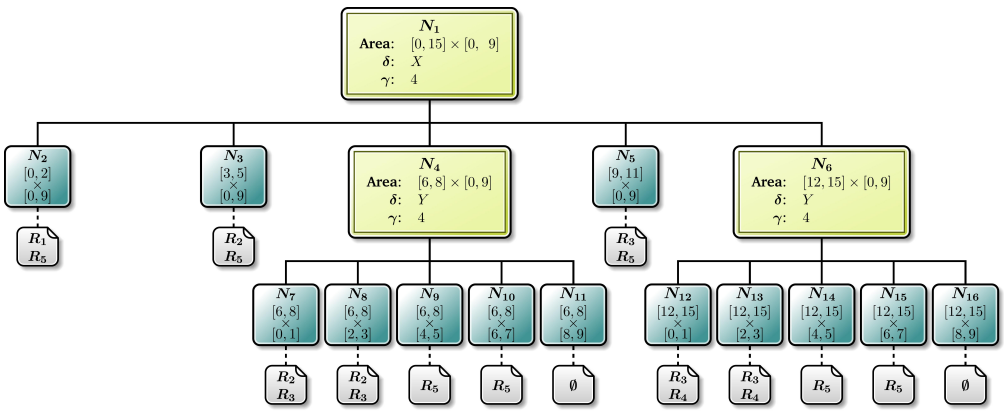


Fig. 4.19: The HiCuts tree for the rule set shown in Figure 4.18.

rules, the resulting decision tree in Figure 4.19 has a considerable size of 16 nodes with a total of 13 sub rule sets associated to the leaf rules. Furthermore, note that the rules R_2, R_3, R_4, and R_5 have a duplication factor of 3, 5, 2, and 7, respectively, which significantly contributes to the tree's size. In order to mitigate the large storage requirements for the decision tree, the HiCuts paper [63] suggests to join neighbored child nodes with the same set of associated rules. For example, in Figure 4.19, the nodes N_7 and N_8 can be joined to a single node $N_{7,8}$ that covers the area $[6,8] \times [0,3]$. Note however, that the number of pointers in the parent node remains the same, and pointers to originally distinct nodes are simply redirected to the joined node.

However, the generated decision tree $T_{\mathfrak{R}}$ for a rule set $\mathfrak{R}$ can be used for quick packet classifications by traversing the tree based on the header data h_p for an incoming packet p, until a leaf node is reached. The traversal starts at the root node and, whenever a non-leaf node N is encountered, the metadata stored in N is used to quickly decide into which child the search must descend. More precisely, when we assume that pointers to the child nodes are stored in a zero-indexed array A_N, then the $\mathcal{O}(1)$ operation

$$A_N[i] \text{ with } i = \min\left\{ \left\lfloor \overbrace{h_\delta - a_\delta^N}^{\text{Header offset}} \Big/ \underbrace{\left\lfloor \frac{b_\delta^N - a_\delta^N + 1}{\delta + 1} \right\rfloor}_{\text{Number of points in child intervals, can be precomputed}} \right\rfloor, \delta \right\} \tag{4.19}$$

leads to the correct child node C_{i+1}. When the search has finally descended into a leaf node N, the corresponding sub rule set $\mathfrak{R}_N$ is searched linearly for the most highly prioritized matching rule. For example, the packet p with header $h_p = (13, 5)$ is classified by starting the tree traversal at the root node N_1 and computing the child index $\min\{\lfloor \frac{13-0}{\lfloor \frac{15-0+1}{5} \rfloor} \rfloor, 4\} = 4$, which points to the node N_6. Subsequently, the index $\min\{\lfloor \frac{5-0}{\lfloor \frac{9-0+1}{5} \rfloor} \rfloor, 4\} = 2$ points to the leaf node N_{14}, whose associated sub rule set $\mathfrak{R}_{N_{14}} = \langle R_5 \rangle$ yields the most highly prioritized matching rule R_5.

With $T_{\mathfrak{R}}^{\uparrow}$ being the height of the decision tree, the overall classification time is in $\mathcal{O}(T_{\mathfrak{R}}^{\uparrow})$, since every classification requires the traversal from the root to a leaf node associated to a rule set with a small maximum size of β. Also, the required space and the tree construction time are proportional to the number of nodes $|T_{\mathfrak{R}}|$ in the decision tree $T_{\mathfrak{R}}$ and therefore in $\mathcal{O}(|T_{\mathfrak{R}}|)$. Both the height $\text{height}(T_{\mathfrak{R}})$ and the number of nodes $|T_{\mathfrak{R}}|$ are hard to predict due to the utilized heuristics. Although it is generally possible to incrementally update a decision tree [63], a rule insertion

Classification operation	Data structure creation	Data structure update	Memory requirements
$\mathcal{O}(T^{\uparrow})$	$\mathcal{O}(\|T\|)$	$\mathcal{O}(\|T\|)$	$\mathcal{O}(\|T\|)$

T: the HiCuts decision tree $T^{\uparrow}$: height of T $|T|$: number of nodes in T

Tab. 4.8: HiCuts performance characteristics.

or deletion can affect every single node in the tree. Furthermore, depending on the nature of the update, such as a rule insertion, additional backtracking may be required in order to avoid deterioration of the tree's structure. According to the literature [58, 61, 107, 144, 147], $|T|$ is in $\mathcal{O}(n^d)$ and $T^{\uparrow}$ is in $\mathcal{O}(d)$. However, these publications do not provide a proof for the abovementioned bounds. These performance indicators are summarized in Table 4.8.

4.5.2 HyperSplit

The *HyperSplit* decision tree technique [107] is a successor algorithm to HiCuts that aims to improve upon HiCuts in both memory footprint and classification performance. At its heart HyperSplit works similar to HiCuts, although it significantly differs from HiCuts when it comes to the heuristic used for the selection of the cut dimension. Furthermore, HyperSplit always performs exactly one cut in the chosen cut dimension, in contrast to the γ cuts performed by HiCuts. Accordingly, the HyperSplit data structure is always a binary tree, which can be stored and accessed efficiently.

The HyperSplit paper [107] describes two different heuristics to choose the dimension to cut, but recommends the usage of one specific heuristic due to its superiority over the other in terms of space savings. Before we dive into the details of the superior heuristic, we introduce the notion of *interval weights*. As we have seen in Section 4.3.1, the endpoints of the geometric rule representations in a dimension j can be used to partition the jth axis of a d-dimensional box. More precisely, when we consider a rule set $\mathfrak{R} = \langle R_1, \ldots, R_n \rangle$, where the geometric representation of each rule $B(R_i) = [a_1^i, b_1^i] \times \ldots [a_d^i, b_d^i]$ lies within a d-dimensional box $B = [a_1, b_1] \times \ldots \times [a_d, b_d]$, the endpoints a_j^i and b_j^i partition the jth axis $[a_j, b_j]$ of B into at most $l_j \leq 2n + 1$ disjoint intervals $I_k^j, k \in \{1, \ldots, l_j\}$. The *weight* $w(I_k^j)$ of an interval I_k^j is the number of rules that intersect with I_k^j in the jth dimension, i. e.,

$$w(I_k^j) = \left| \left\{ R_i \middle| \left[a_j^i, b_j^i \right] \cap I_k^j \right\} \right|, \tag{4.20}$$

as depicted in Figure 4.20 for two-dimensional rules. In order to pick a suitable

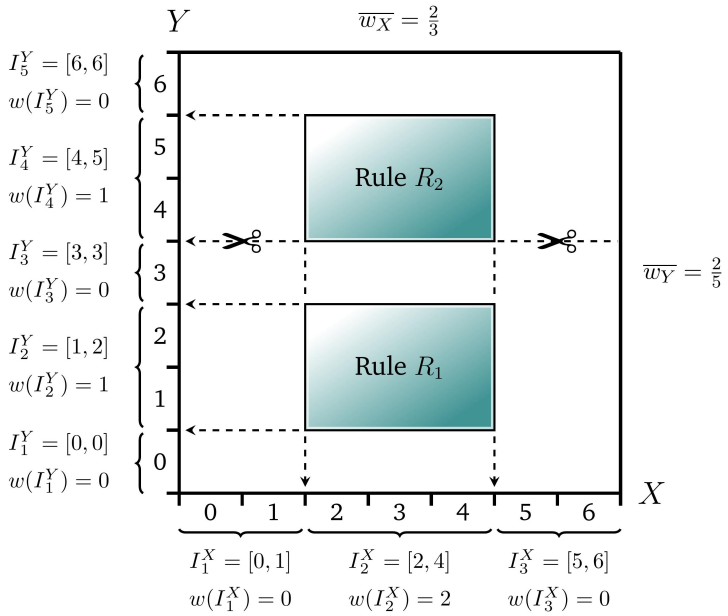


Fig. 4.20: HyperSplit cut dimension and cut point determination using interval weights.

dimension δ to cut, HyperSplit's dimension heuristic chooses the dimension δ with the smallest average interval weight $\overline{w_\delta}$, with

$$\overline{w_j} = \sum_{k=1}^{l_j} w(I_k^j) \Big/ l_j. \tag{4.21}$$

Since the cut will take place in one interval I_k^δ, possibly every rule that intersects with I_k^δ will be duplicated. Therefore, the heuristic's idea is to pick the dimension where the average number of possibly duplicated rules is minimal. Accordingly, the dimension Y would be chosen to cut in Figure 4.20, since $\overline{w_Y} < \overline{w_X}$.

The next important step is to determine the cut point ρ in the interval $[a_\delta, b_\delta]$. A reasonable choice for ρ would be a point that separates half of the rules in B into the interval $I_{\text{left}} = [a_\delta, \rho - 1]$ and the other half into $I_{\text{right}} = [\rho, b_\delta]$. Although this is not always possible, e. g., due to rule overlaps, HyperSplit aims to approximate such a bisection by choosing ρ such that the interval weights of I_{left} and I_{right} are close. To this end, ρ is set to the start point of the interval I_m^δ, such that m is the smallest value in $\{1, \ldots, l_\delta\}$ with

$$\sum_{k=1}^{m} w(I_k^\delta) > \frac{1}{2} \sum_{k=1}^{l_j} w(I_k^\delta). \tag{4.22}$$

In Figure 4.20, ρ would be set to $m = 4$, because $m = 4$ is the smallest value that satisfies Condition 4.22. The figure indicates that this cut point indeed separates rule R_1 from rule R_2.

The general procedure that generates a HyperSplit decision tree from a specified rule set is analog to HiCuts. Figures 4.22 shows the HyperSplit tree that would be generated from the rule set given in Figure 4.21. It becomes apparent that the HyperSplit tree in Figure 4.22 has the same height as the HiCuts tree from Figure 4.19, but requires only five nodes (in contrast to the 16 nodes of the HiCuts tree). The reason for this is the greater care with which HyperSplit places its cuts with regard to tree balancing and rule duplication avoidance.

The worst case complexity of HyperSplit does not differ from that of HiCuts in terms of tree height $T^{\uparrow}$ and number of nodes $|T|$ in the tree. However, the authors

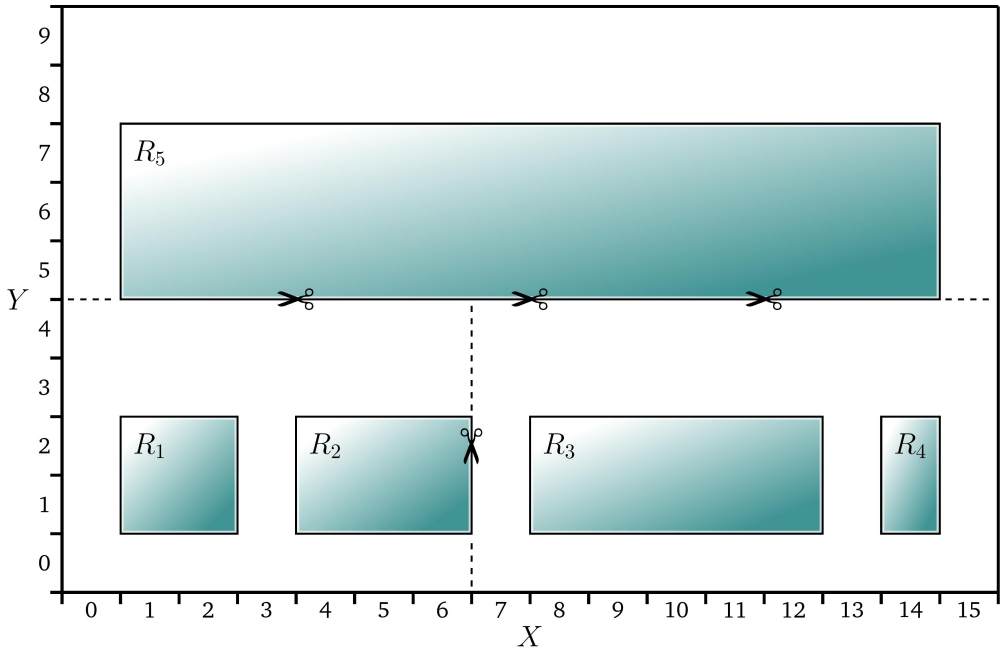


Fig. 4.21: Complete HyperSplit example with five rules, $\beta = 2$, and $\mathcal{H} = [0, 15] \times [0, 9]$.

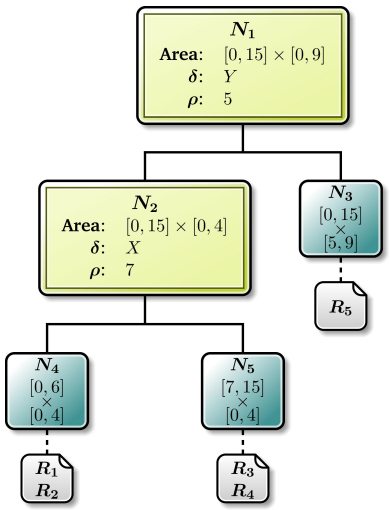


Fig. 4.22: The HyperSplit tree for the rule set shown in Figure 4.21.

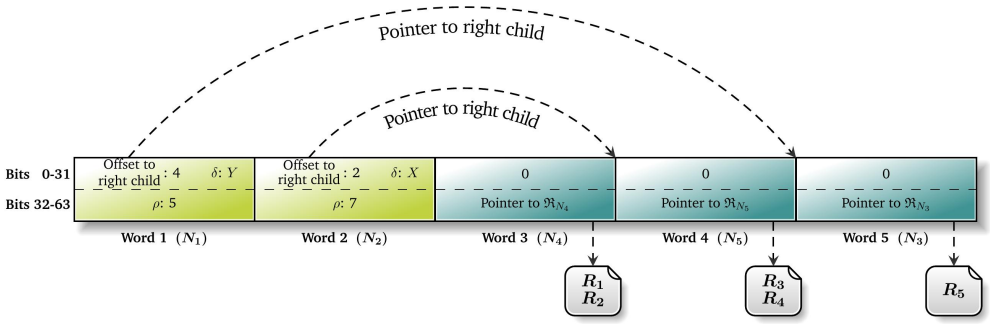


Fig. 4.23: Array representation of the HyperSplit tree shown in Figure 4.22.

of the HyperSplit paper suggest that $T^{\uparrow} \in \mathcal{O}(d \cdot \log(2n+1))$ [107]. In comparison to HiCuts trees, however, the binary HyperSplit trees have the advantage that they can be stored and accessed in a compact way using an array representation, as sketched in Figure 4.23. Here, each node in the HyperSplit tree is represented as a 64 bit word. In case the word represents an inner node, the first 32 bit contain the offset to the word representing the right child node (the left child node is always the rightmost next word) and the cut dimension δ, while the second 32 bit store the cut point ρ. If the word represents a leaf node, the first 32 bit are zero, while the second 32 contain a pointer to the corresponding sub rule set. That way, the entire search data structure can not only be stored in one coherent memory chunk (except for the sub rule sets), it also requires only forward jumps to be executed, which allows for cache-efficient memory accesses.

4.6 Virtual Machines

The related work we have discussed up to this point is mainly concerned with algorithmic techniques to solve the GPCP. Besides the abovementioned classification algorithms, a widely used and implemented methodology within a UNIX/Linux Operating System (OS) is to rely on a *Domain Specific Language (DSL)* for network packet processing, which is executed in a corresponding *VM* [101]. Here, a rule set, as defined in Section 2.2.2, or a more complex packet processing policy are specified as a sequence of instructions that describe how packets or other messages are to be treated at specific hooks within the OS [99]. Although these VM approaches are technically orthogonal to the dedicated classification algorithms we are concerned with, we briefly introduce the most important works in this direction, as they represent a milestone in the (current) history of UNIX/Linux network packet processing.

One of the earliest implementations of a packet processing VM is provided by Mogul, Rashid, and Accetta, which is simply titled *The Packet Filter* [101]. The Packet Filter is implemented as a stack-machine-based interpreter residing in the OS kernel, that can execute bytecodes generated from small assembly-level programs each time a packet needs to be processed. One of The Packet Filter's main features is its ability to update the packet processing programs from within user processes, without the need to recompile the kernel code. However, the authors also mention potential performance issues implied by the fact that the packet processing programs are interpreted rather than natively compiled.

In [94], McCanne and Jacobson present the *Berkeley Packet Filter (BPF)*, a spiritual successor to The Packet Filter, which is conceptually similar, but replaces the stack-based interpreter by a register machine due to performance improvements. Since its inception, BPF-like techniques have gained considerable attention by the networking community and are nowadays used as the standard (or optional) backend packet processing engine in tools such as `tcpdump` [173] or `iptables` [166]. Further works in the BPF domain improve the execution performance of the packet processing programs by introducing a Just-in-time Compiler (JIT) in order to remove the interpretation overhead [33, 53]. Furthermore, [53] and [33] propose the utilization of several optimization techniques known from compiler theory, such as peephole optimization [129], constant folding [145], and further optimizations based on an Static Single Assignment (SSA) [28] representation of filter programs.

The most recent and widely used installment in the BPF VM series is the *extended Berkeley Packet Filter (eBPF)* [171], which not only allows for packet processing, but also for monitoring tasks [99] and ACLing system calls. Furthermore, eBPF provides reasonably high-level data structures such as maps, which can be used for the implementation of efficient classification algorithms, such as the in Section 4.3 described Bit Vector Search [34].

5 JitVector with SIMD Instructions

The Bit Vector (BV) Search algorithm, as introduced in Section 4.3, is among the fastest existing classification algorithms that solve the Geometric Packet Classification Problem. Despite its $\mathcal{O}(n)$ worst case classification operation, BV Search outperforms dynamic approaches such as Linear Search or Tuple Space Search. Furthermore, BV search is on par with or even faster than decision tree algorithms (see Section 4.5), at more predictable (and mostly lower) memory footprints and preprocessing times [10]—in fact, a severe problem from which many high performance classification techniques suffer are large memory requirements and long preprocessing times to compute the search data structures, especially when the rule sets grow in size [34, 10]. This is also true for the RFC approach [62], which provides constant lookup time at exponential memory and preprocessing time requirements, as explained in Section 4.4.1.

In this chapter, we present three techniques to improve the performance of the compute-intensive stages of the BV algorithm, namely the binary searches and the vectorized ANDs. First, for large dimensions with more than 16 header bits, we directly embed the binary search tree into the instruction stream by just-in-time compiling the binary search tree into a compact opcode representation that can be traversed without additional data loads, backward jumps, and division operations. Second, we replace the binary search for small dimensions with up to 16 bits by lookup tables that directly map header fields to the corresponding bit vectors. Third, we utilize machine-specific SIMD operations to accelerate the linear part of the BV technique's operations, namely the aggregation of the one-dimensional result vectors and the location of the first set bit.

Our results demonstrate a significant performance gain for the BV approach up to factor of 2.8$\times$ in terms of lookup speed, at the cost of moderate increases in memory consumption and preprocessing time. We also apply our adaptations to the Aggregated Bit Vector (ABV) algorithm, whose performance can be improved by up to a factor of 1.5$\times$.

5.1 Accelerating One-dimensional Searches

As explained in Section 4.3.1, the BV Search classification process executes two consecutive execution phases to classify an incoming packet p with the header fields $h^p = (h_1^p, \ldots, h_d^p)$: d one-dimensional searches and combination of the one-dimensional results. In the first phase, the BV algorithm needs to locate the intervals $I_{k_j}^j$, such that

$$\forall j \in \{1, \ldots, d\} : h_j^p \in I_{k_j}^j. \tag{5.1}$$

The original publication [76] explicitly suggests to employ binary search to locate the intervals $I_{k_j}^j$, which leaves the first BV search phase with

$$\mathcal{O}(d \cdot \log(n)) \tag{5.2}$$

execution steps. Without loss of generality, we say that a dimension j is *large* iff $|H_j| > 2^{16}$, otherwise we say that the dimension j is *small*.

5.1.1 Direct Lookups for Small Dimensions

During the BV Search data structure creation, a header field domain H_j is partitioned into δ_j intervals, with

$$1 \leq \delta_j \leq \min\left(2 \cdot n + 1, 2^{Y_j}\right). \tag{5.3}$$

Accordingly, in dimension j, a binary search within these intervals requires $\mathcal{O}(\log(\delta_j))$ many steps to complete.

Our first acceleration approach is inspired by the first stage of the RFC approach [62] and exploits the fact that certain network packet header field domains H_j require only few bits Y^j for representation. For example, in IP packets, TCP or UDP port fields are represented with sixteen bits, while the protocol (for IPv4 packets) and the next header (for Internet Protocol Version 6 (IPv6) packets) are encoded in only eight bits. These domains are small enough to allow for an explicit interval table representation, which can be looked up directly using the corresponding header field. Figures 5.1 and 5.2 provide an example for a one-dimensional rule set in a small dimension, for which such a lookup table is computed.

Although such an interval lookup table requires significantly more memory than simply storing the interval end points for binary search purposes, it enables $\mathcal{O}(1)$

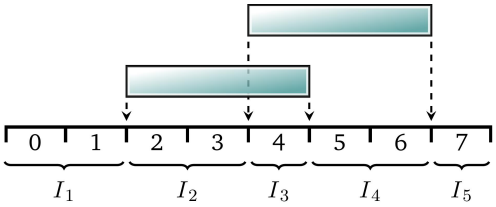


Fig. 5.1: Intervals for a rule set in a three-bit dimension.

Header value	0	1	2	3	4	5	6	7
Interval index	1	1	2	2	3	4	4	5

Fig. 5.2: The interval lookup table corresponding to Figure 5.1.

access to the desired interval index k_j. In comparison to RFC, however, we only apply this technique to small dimensions, which leads to a comparatively small memory overhead. Moreover, in contrast to RFC, this overhead is constant for all rule set sizes, because we do not compute table crossproducts based on equivalence IDs.

5.1.2 JITing Binary Searches in Large Dimensions

As discussed in the previous section, lookup tables for quick interval header-to-interval mappings are only practical for small dimensions j with relatively small header field domains H_j, as the number of table entries grows exponentially with $|H_j|$. Therefore, as suggested in [76], binary searching the intervals of large dimensions is preferable to direct table lookups, since storing the interval delimiters is significantly less memory intensive. Generally, a binary search in dimension j can be represented as a function

$$\mathrm{bs}^j : H_j \times \mathbb{P}\left(H_j\right) \to \mathbb{N}_0, \tag{5.4}$$

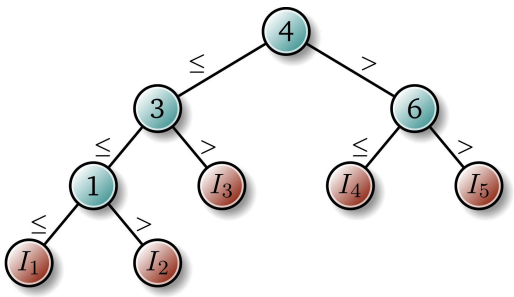


Fig. 5.3: The intermediate binary search tree that is to be JIT-compiled into machine code.

that uses a header field in H_j to search in a list of interval delimiters, which is an element of the power set of H_j. In the remainder of this section, we refer to the interval delimiters in dimension j by δ_j.

In order to accelerate the binary search in a large dimension j, we propose to specialize the binary function bs^j to an unary function $\text{bs}^j_{\delta_j}$ by partially evaluating the binary search with respect to the interval delimiters δ_j. That is, for each large dimension j, we generate a specialized binary search by embedding the interval delimiters as well as the corresponding comparison instructions directly into the instruction stream that is executed whenever a packet needs to be classified. We do this by JIT-compiling specialized binary search functions

$$\text{bs}^j_{\delta_j} : H_j \to \mathbb{N}_0 \tag{5.5}$$

for every large dimension j when a rule set $\mathfrak{R}$ is loaded. To this end, we add two additional preprocessing steps to the standard BV Search data structure creation operation: first, after the interval delimiters have been collected, we generate a corresponding binary search tree structure. Second, the binary search tree is traversed in pre-order to generate the machine instructions that directly implement this specific search tree. The generated machine code inlines the tree's left branches and does neither require addition nor shift operations. Furthermore, data memory loads due to the embedded constants are avoided.

At run time, whenever a header field h_j needs to be mapped to an interval $I^j_{k_j}$ in a large dimension j, we call the specialized binary search function $\text{bs}^j_{\delta_j}$, i. e.,

$$k_j \leftarrow \text{bs}^j_{\delta_j}(h_j)\,. \tag{5.6}$$

In order to allow for branchless iteration over all dimensions (i. e., small and large ones), the table lookups in small dimensions are also JIT-compiled. That way, we can simply store a function pointer for each dimension that is called during a classification operation, without the need to test whether a dimension is large or small in every loop iteration over the dimensions, as shown in Listing 5.2.

```
1       cmp rdi, 0x00000004
2       ja  L1
3       cmp rdi, 0x00000003
4       ja  L3
5       cmp rdi, 0x00000001
6       ja  L4
7       mov rax, 0x00000000
8       ret
9   L4:
10      mov rax, 0x00000001
11      ret
12  L3:
13      mov rax, 0x00000002
14      ret
15  L1:
16      cmp rdi, 0x00000006
17      ja L2
18      mov rax, 0x00000003
19      ret
20  L2:
21      mov rax, 0x00000004
22      ret
```

Listing 5.1: Assembly code for the JIT-compiled binary search function that corresponds to the search tree in Figure 5.3, with inlined left branches. Note that the interval indices are zero-based.

5.2 Accelerating the Aggregation Phase

We now turn our attention to the acceleration of the BV Search aggregation phase. Having located the indices k_j of the intervals $I_{k_j}^j$ during the one-dimensional searches, the BV algorithm retrieves the corresponding bit vectors $V_{k_j}^j$. For each dimension j, the vector bit $V_{k_j}^j[i]$ is set to one if the rule R_i matches the header field h_j^p, otherwise it is set to zero. As such, the position i^* of the first set bit in the vector

$$V_{\text{res}} = \bigwedge_{j=1}^{d} V_{k_j}^j \tag{5.7}$$

provides the index of the most highly prioritized matching rule R_{i^*}. In the original BV publication [76], the authors suggest to store the vectors in chunks of width w, the system's memory word width. They also argue that the worst-case execution time decreases with increasing w. As such, the BV aggregation phase takes

$$\mathcal{O}\left(d \cdot \left\lceil \frac{n}{w} \right\rceil\right) \tag{5.8}$$

time, with w being typically 32 or 64 on currently used CPUs. Listing 5.3 shows our C implementation of the BV Search aggregation phase using 64 bit words.

```
1  size_t
2  JVStorage_classify(
3      JVStorage* storage,
4      Header* header) {
5
6    VectorIntervalStorage* vi_storage = storage->vi_storage_;
7    const size_t num_dims = vi_storage->num_dims_;
8    BitVector** vectors = storage->tmp_vectors_;
9
10   // Gather the vectors for each dimension by calling
11   // JIT-compiled functions.
12   for (size_t i = 0; i < num_dims; ++i) {
13     const FieldVal field = header->fields_[i];
14     BitVector** dim_vectors = vi_storage->vectors_[i];
15
16     // Execute interval lookup using a JIT-compiled function.
17     // We do not need to branch here, because the table lookups
18     // for small dimensions are also JIT-compiled.
19     JitProc_t bin_search_func = storage->bin_searches_[i];
20     const size_t index = bin_search_func(field);
21     vectors[i] = dim_vectors[index];
22   }
23   // Now we look for the first set bit in the aggregation phase.
24   return BitVector_aggregation_phase(vectors, num_dims);
25 }
26 }
```

Listing 5.2: Our C implementation of JITed one-dimensional searches.

Current Intel and AMD CPUs provide support for Streaming SIMD Instructions (SSE) and Advanced Vector Extensions (2) (AVX(2)) instructions, which operate on 128 and 256 bit registers, respectively. More recent Intel CPUs also implement the Advanced Vector Extensions 512 (AVX-512) instruction set, thereby further extending SIMD register sizes to 512 bits [169]. We now aim to improve the performance of the machine-word bit vector aggregation phase by utilizing SIMD instructions to AND the vectors and to retrieve the position of the first set bit. As such, we need to provide vectorized formulations for Step 1 and Step 2 from Listing 5.3, respectively.

Each of the abovementioned instruction sets provides direct implementations for loading and ANDing W-bit words, with W being 128 for SSE, 256 for AVX(2), and 512 for AVX-512, respectively. Without loss of generality, we only describe the BV Search aggregation step for AVX-512 in the remainder of this section, as ports to SSE and AVX(2) are straightforward. For example, AVX-512 defines the `vmovdqa64` and `vpandq` instructions for loading and ANDing. As such, Step 1 can be directly ported to the corresponding SIMD variant, as demonstrated in Listing 5.4 in lines 11 to 19 for AVX-512. Note that we use Intel intrinsic functions [170] instead of inline assembly code for readability reasons.

```
1  size_t
2  BitVector_aggregation_phase(
3      const BitVector** vectors,
4      const size_t num_vectors) {
5
6    const size_t num_words = vectors[0]->num_words_;
7    // Loop over the 64-bit vector words.
8    for (size_t i = 0; i < num_words; ++i) {
9      // Step 1: ANDing the vector words.
10     uint64_t word = vectors[0]->words_[i];
11     for (size_t j = 1; j < num_vectors; ++j) {
12       word &= vectors[j]->words_[i];
13     }
14
15     // Step 2: Finding the first set bit.
16     // If the ANDed word is greater than zero,
17     // at least one bit is set.
18     if (word > 0) {
19       // Return the offset of the set bit.
20       // __builtin_ctzll is a gcc builtin for the
21       // `tzcnt` instruction which counts the number
22       // of trailing zero bits.
23       // Our implementation stores the most highly
24       // prioritized bit at the least significant position.
25       return (1 + i * 64 + __builtin_ctzll(word));
26     }
27   }
28   // A return value of 0 indicates no match.
29   return 0;
30 }
```

Listing 5.3: Our C implementation of the BV Search aggregation phase using 64 bit words.

Step 2, the extraction of the first set bit's position from the ANDed result vector, is slightly more involved because of two reasons: first, except for SSE, there exists no instruction to test whether an entire W-bit word x is zero, and second, the instruction sets also lack an instruction to find the index of the most highly prioritized set bit. Therefore, we perform a vectorized comparison of x against 32-bit zeros, which yields a 16-bit result mask m, as shown in lines 23 to 27 in Listing 5.4. Here, a mask bit $m[i]$ is set iff $x[32 \cdot i + 31 \,..\, 32 \cdot i] \neq 0$. Also, we store x on the stack in order to achieve subword access (lines 33 to 34). If $m \neq 0$, at least one 32-bit word inside x contains a set bit, otherwise we continue to the next 512-bit vector word. The most highly prioritized set bit in x is extracted as follows: first, we find the most highly prioritized 32-bit subword u in x with $s \neq 0$ by finding the index of the most highly prioritized set bit in m (lines 36 to 38). Second, we extract the position of the most highly prioritized bit in u. The final desired index i^* can now be computed by adding the bit offsets of the word x, the subword u, and index of the most highly prioritized bit in u (line 43).

```
1  size_t
2  BitVector_aggregation_phase_512(
3      const BitVector** vectors,
4      const size_t num_vectors) {
5
6    const size_t num_words = vectors[0]->num_words_;
7    const size_t num_vector_words = num_words / 8;
8    // Loop over the 512-bit vector words.
9    for (size_t i = 0; i < num_vector_words; ++i) {
10     // Step 1: ANDing the vector words.
11     const size_t addr_offset = i * 8;
12     const uint64_t* addr = vectors[0]->words_ + addr_offset;
13     __m512i vector_word = _mm512_load_epi64(addr);
14     for (size_t j = 1; j < num_vectors; ++j) {
15       vector_word = _mm512_and_epi64(
16         vector_word,
17         _mm512_load_epi64(vectors[j]->words_ + addr_offset)
18       );
19     }
20
21     // Step 2: Finding the first set bit.
22     // 32-bit-wise vectorized comparison with zero.
23     #define NOT_EQUAL_OP 4
24     const uint16_t mask = _mm512_cmp_epi32_mask(vector_word,
25                                                 _mm512_setzero_si512(),
26                                                 NOT_EQUAL_OP);
27     #undef NOT_EQUAL_OP
28     // If the mask is not zero, at least one bit is set in the ANDed
29     // 512-bit vector word.
30     if (mask != 0) {
31       // Store the ANDed 512-bit vector word as consecutive
32       // 32-bit words.
33       uint32_t words[16];
34       _mm512_store_si512(words, vector_word);
35       // Find the first 32-bit word that is not zero.
36       const size_t word_index = __builtin_ctz(mask);
37       const size_t word_offset = 32 * word_index;
38       const uint32_t word = words[word_index];
39       // Find the index of the first set bit in the 32-bit word.
40       const size_t bit_offset = __builtin_ctz(word);
41       const size_t avx_word_offset = i * 512;
42       // Return the position of the first set bit in the vector.
43       return (1 + avx_word_offset + word_offset + bit_offset);
44     }
45   }
46   // A return value of 0 indicates no match.
47   return 0;
48 }
```

Listing 5.4: Our C implementation of the BV Search aggregation phase using 512 bit words, using Intel intrinsic functions.

Using this methodology, the number of iterations in the outer loop is reduced by factors of 2 for SSE, 4 for AVX(2), and 8 for AVX-512, which can significantly increase the performance of the linear runtime component in the BV Search approach.

Approach	Classification operation	Data structure creation	Data structure update	Memory requirements
		Related work		
Bit Vector Search [76]	$\mathcal{O}\left(d \cdot \log(n) + d\left\lceil \frac{n}{w} \right\rceil\right)$	$\mathcal{O}(dn^2)$	$\mathcal{O}(dn^2)$	$\mathcal{O}(dn^2)$
		Proposed approach		
Jit Vector Search	$\mathcal{O}\left((d-d_s)\log(n) + d_s + d\left\lceil \frac{n}{W} \right\rceil\right)$	$\mathcal{O}\left(dn^2 + 2^{Y_s} d_s\right)$	$\mathcal{O}\left(dn^2 + 2^{Y_s} d_s\right)$	$\mathcal{O}\left(dn^2 + 2^{Y_s} d_s\right)$

n: number of rules d: number of dimensions w: machine word width W: SIMD word width
d_s: number of small dimensions Y_s: maximum number of bits in small dimensions

Tab. 5.1: Jit Vector Search performance characteristics, in comparison to related work.

5.3 Performance Characteristics

When compared to the original BV Search algorithm, the Jit Vector Search approach provides increased classification performance due to quick table lookups in small dimensions, specialized binary searches in large dimensions, and SIMD-based vector operations during the aggregation phase. These performance improvements come at the cost of a moderate increase in the required memory footprint as well as the data structure preprocessing time, due to the complete lookup table computation and machine code generation steps. The usage of SIMD instructions comes at little additional memory overhead, because bit vectors need to be zero-padded to

$$\left\lceil \frac{n}{W} \right\rceil \cdot W \tag{5.9}$$

many bits in order to enable special-case-free iteration over the vectors, as shown in the outer loop in Listing 5.4. Table 5.1 summarizes the key performance indicators of the Jit Vector Search in comparison to BV Search. The table entries also apply for the ABV Search approach, which offers the same worst-case complexities.

5.4 Evaluation

Having discussed the theoretical performance characteristics of the proposed Jit Vector Search in Section 5.3, we now present the results of our practical evaluation. During the evaluation, we focus on three key performance indicators, namely *classification speed*, *search data structure memory consumption*, and *search data structure preprocessing time*. We evaluate these characteristics with respect to the Geometric Packet Classification Problem, as introduced in Section 2.3, as the

GPCP provides the most common ground for the different algorithmic techniques under scrutiny. The goal of this section is to demonstrate the viability of Jit Vector Search in terms of the abovementioned practical metrics, when compared to existing basic as well as state-of-the-art classification approaches.

5.4.1 Experiment Setup

In order to evaluate the different variants of Jit Vector Search and as well as existing work, we use the tool *ClassBench* [132] to generate rule sets and packet header traces. For each rule set size in

$$\{2^i | i \in \{4, 6, 8, 10, 12, 14, 16\}\}, \tag{5.10}$$

we generate 144 different synthetic five-dimensional geometric rule sets, each with a corresponding header trace of 50,000 headers. The five packet header dimensions are source and destination IPv4 address fields, source and destination port fields, as well as the layer four protocol field. The packet headers in every trace are uniformly distributed over the rules in the corresponding rule set, such that each rule approximately matches the same amount of headers. Furthermore, every packet header always matches at least one rule, i. e., there are no packet headers in a trace which do not match at least a single rule in the corresponding rule set. As such, our evaluation data consists of 1,008 different synthetic rule sets alongside their corresponding traces.

The classification algorithms used during the evaluation are the proposed (Aggregated) Jit Vector Searches for SSE, AVX(2), and AVX-512 instructions set enhancements, alongside the existing (Aggregated) Bit Vector Searches [31, 76], the RFC algorithm [62], Tuple Space Search [127], the HiCuts [63] and HyperSplit [107] decision tree approaches, and Linear Search. We do not include On Demand Crossproducting (ODC) [128] in the list of evaluated existing approaches, because ODC, in contrast to all other regarded algorithms, heavily relies on temporal locality of the processed traces. This trait, however, is not provided by our experiment setup, which negatively biases ODC's classification performance. As such, we argue that the ODC classification results from our experiment are not representative and are therefore not shown.

For each (Algorithm A, rule set $\mathfrak{R}$ with corresponding trace T) combination, we conduct the following experiment: first, A's search data structure S_A is computed, the required build time and memory footprint are recorded. Second, the search data structure S_A is used to classify each header in the trace T, the classification time as well as the match indices are recorded. Finally, the recorded match indices

are compared against a precomputed Linear Search result in order to ensure correctness. The data structure construction, the header classifications, and the result verification are executed in a userspace tool written in C, which is compiled using `gcc 7.4` with the following flags:

```
-Wall -Werror -Wextra -pedantic-errors -std=gnu99 -march=native -O2.
```

The rule sets and header traces are read from simple text files, such that no real network access and packet parsing is required. The evaluation is conducted on an otherwise idle Ubuntu 18.04 Linux system, which runs on an Intel i9-7900X CPU with 64 GB of RAM.

Finally, it is important to note that we must take two precautions in order to prevent algorithm misbehaving: we limit the data structure creation time to at most five minutes, and also, we limit the total amount of allocatable memory for search data structure to four gigabytes. These precautions are essential because the RFC, HiCuts, and HyperSplit algorithms might require several hours of data structure preprocessing due to RFC crossproduct explosions [58, 81] or misbehaving decision tree cut heuristics [83]. If at least one of these limitations is hurt, we consider the algorithm search data structure creation as failed.

5.4.2 Classification Time, Memory Footprint, Preprocessing Time

We begin the review of our algorithm performance evaluation by inspecting the number of observed unsuccessful search data structure creation attempts for the different approaches, as illustrated in Figure 5.4. The figure shows that, with increasing rule set size, the number of forcefully terminated builds increases for the RFC, HiCuts, and HyperSplit algorithms. This is especially well visible for rule sets with 65,536 rules, where 33.3 % of the RFC builds, 84.0 % of the HiCuts builds, and 95.1 % of the HyperSplit builds are unsuccessful. As we can see at the example of HiCuts, this can already happen at relatively small rule set sizes of 256 rules. This behaviour is well known both for decision trees [122, 137] as well as RFC [58, 81] and is considered a major obstacle for practical classification system implementations, such as Open vSwitch [105]. Despite this fact, these approaches, and especially RFC, are amongst the fastest algorithms with respect to classification performance, as we will see in the remainder of this section. We note that all other evaluated algorithms do not suffer from the abovementioned scalability problems.

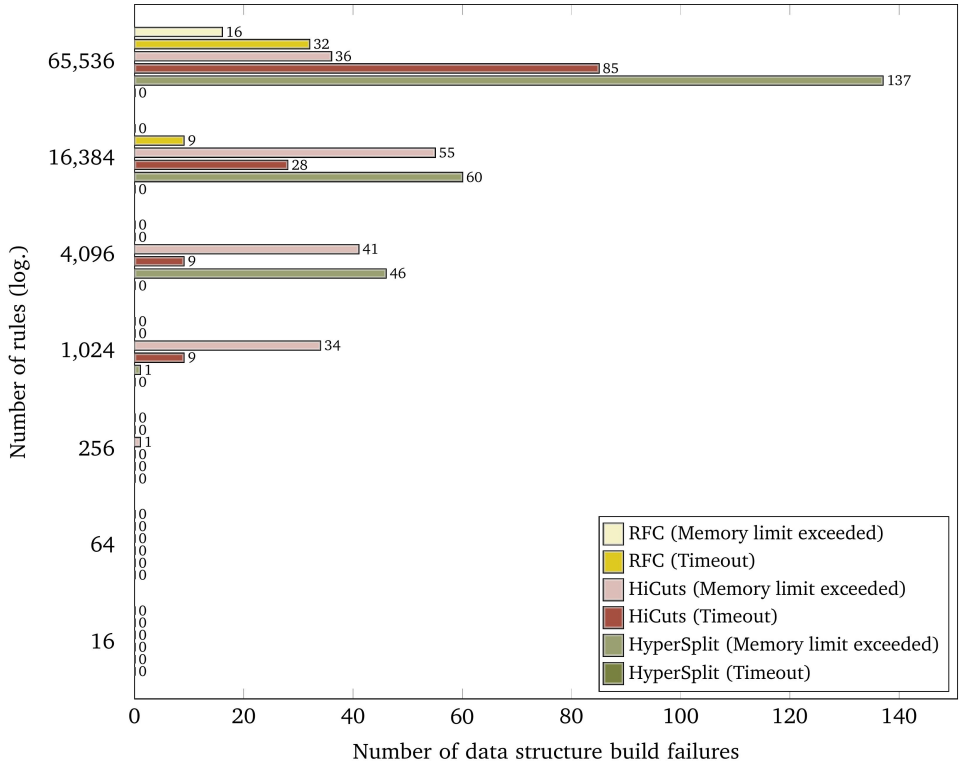


Fig. 5.4: Forcefully terminated algorithm search data structure creation attempts for the RFC, HiCuts, and HyperSplit approaches for different rule set sizes.

Having discussed the scalability issues of existing best-in-class approaches, we now move on the measured average algorithm classification times, memory footprints, and preprocessing times, which are summarized in Figure 5.5, Figure 5.6, and Figure 5.7, respectively. Figure 5.5 shows that the JITing of the binary searches leads to improved classification performance for every regarded rule set size, when applied to the vanilla (Aggregated) Bit Vector Search algorithm. However, the results also imply that the JIT can be considered a micro optimization, as the performance gain is scarcely influenced by the number of rules. The biggest relative performance gain factor of up to 1.5× (for the Bit Vector Search) is achieved for small rule sets with 16 or 32 rules, where the Jit Vector Search algorithm provides the overall best classification performance. The gain in classification performance, however, is bought by a significant increase in memory footprint and preprocessing time, as shown in Figure 5.6 and Figure 5.7, respectively. This overhead is mainly caused by the lookup table computation for small dimensions, which is constant and becomes less significant with an increasing number of rules.

When taking a look at the classification performance of the (Aggregated) Jit Vector Search approaches with SIMD instructions, we notice that for smaller rule set sizes than 1,024 rules, the non-SIMD variants are slightly faster. This is explained by the SIMD way to compute the index of the first bit, which is more complex than

the native 64 bit variant, as well as the SIMD register loads, as shown in Listing 5.3 and Listing 5.4. For larger rule set sizes with at least 1,024 rules, however, we clearly observe a significant performance improvement for the Jit Vector Search up to a factor of 2.6×. Also, as expected, for large rule sets, the performance gain increases with the SIMD instruction bit width W. In case of the Aggregated Jit Vector Search, the performance gain is significantly smaller. The reason for this behaviour is the fact that the Aggregated Bit Vector Search can skip many vector operations due to sparsely populated vectors, as explained in Section 4.3.2. As such, it does not take the same large advantage of SIMD instructions as the non-aggregated variant, as it performs significantly fewer vector operations. Nevertheless, the combination of SIMD instructions and the binary search JIT brings the Aggregated Bit Vector Search close to RFC's performance, without suffering from scalability issues and at significantly faster preprocessing times and lower memory footprints.

When compared to other high-performance algorithms, namely HiCuts and HyperSplit, we see that the fastest (Aggregated) Jit Vector Search variant always beats the decision tree algorithms in terms of classification performance. When it comes to preprocessing time and memory footprint, this is also true for medium to large rule set sizes.

Finally, we take a look at the dynamic Linear Search and Tuple Space Search approaches. While these algorithms provide superior preprocessing performance and low memory footprints, they clearly do not scale for larger rule set sizes with respect to lookup speed. For rule sets with 64 K rules, our fastest Jit Vector approach is about 3,667× faster than Linear Search and about 436× faster than Tuple Space Search. It should be mentioned that also for small rule set sizes, the Bit/Jit Vector Searches clearly outperform Linear Search and Tuple Space Search.

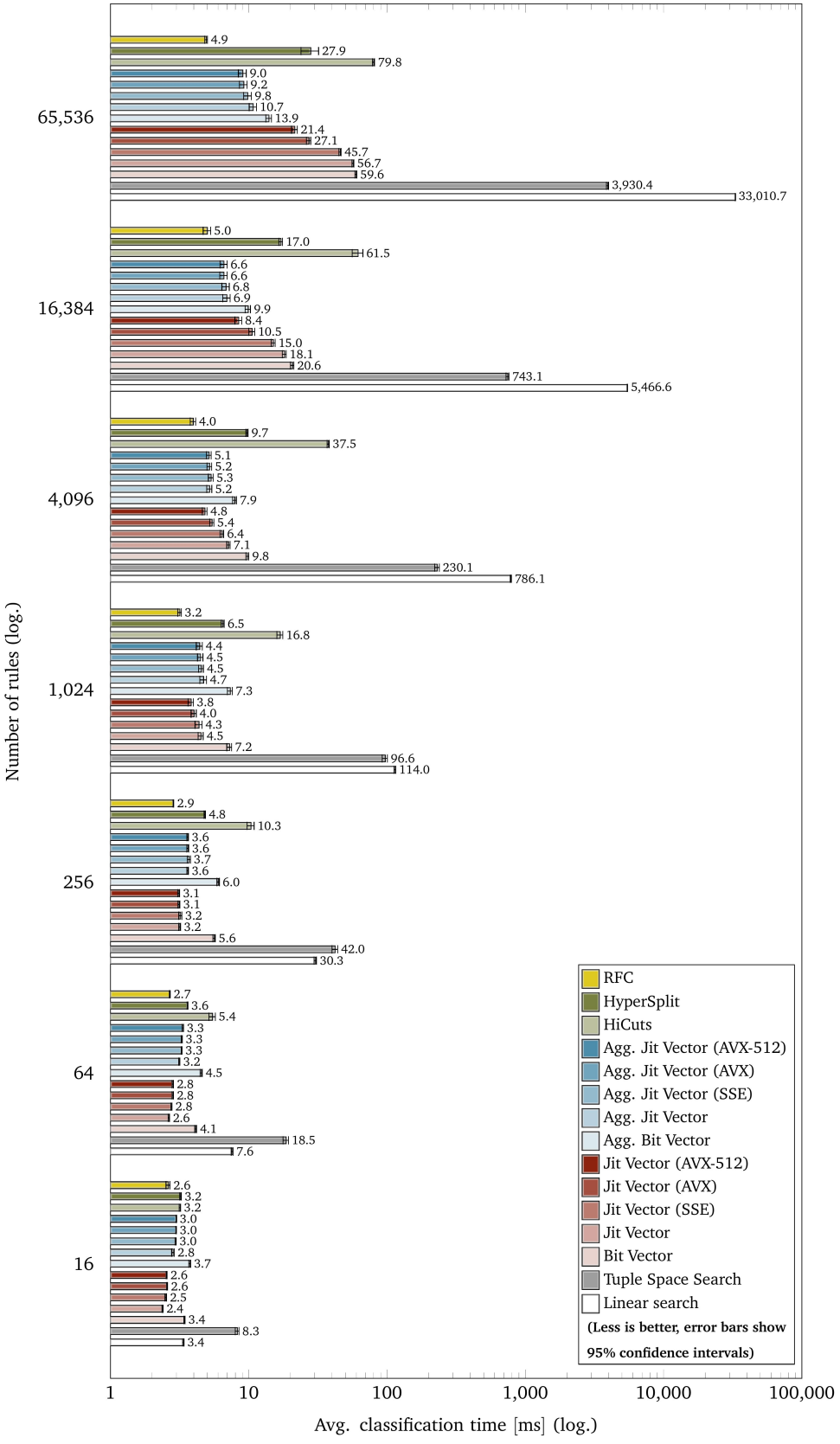


Fig. 5.5: Average algorithm classification times for different rule set sizes required for traces of 50K headers. Note that the RFC, HyperSplit, and HiCuts results only show results of successful search data structure builds.

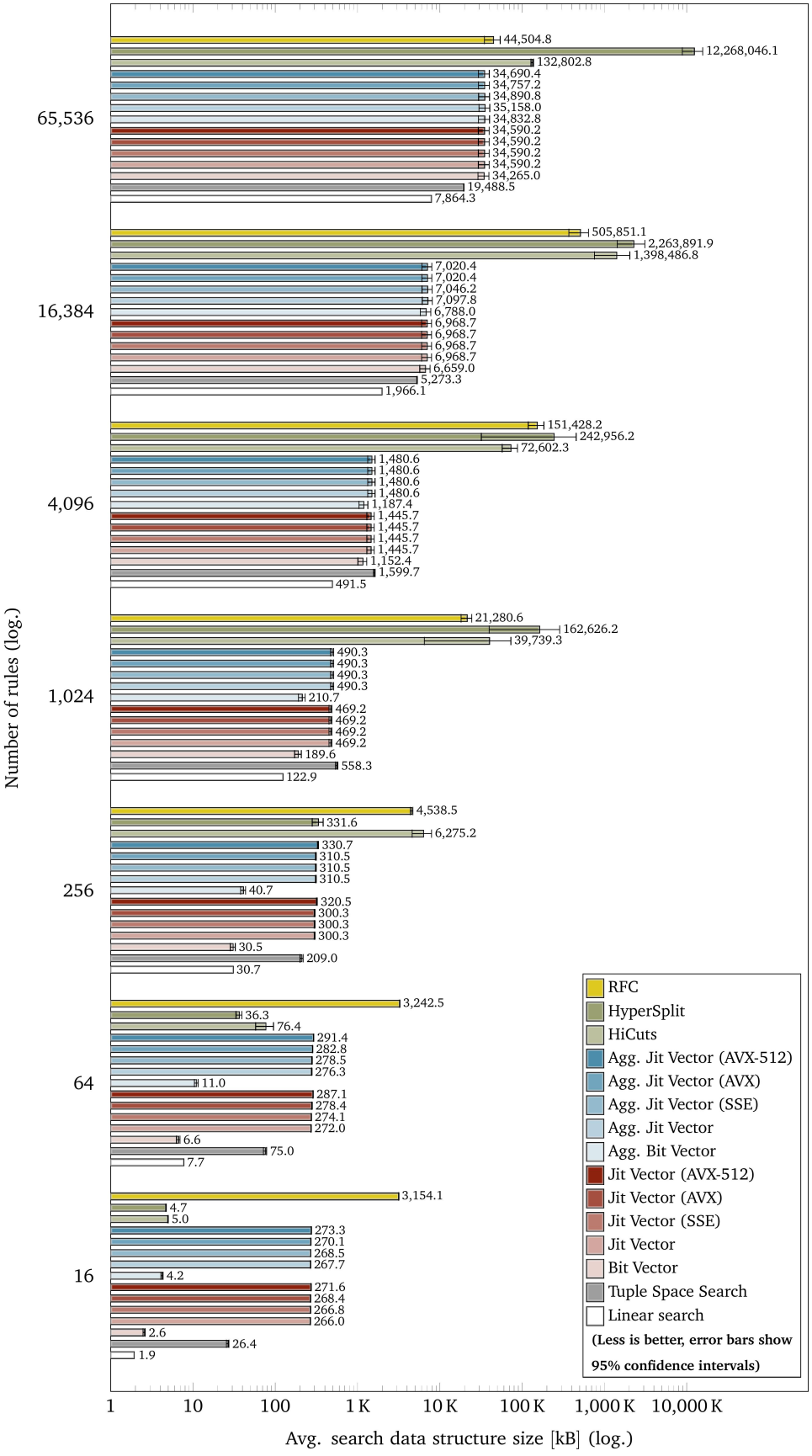


Fig. 5.6: Average memory footprints of algorithm search data structures for different rule set sizes. Note that the RFC, HyperSplit, and HiCuts results only show results of successful search data structure builds.

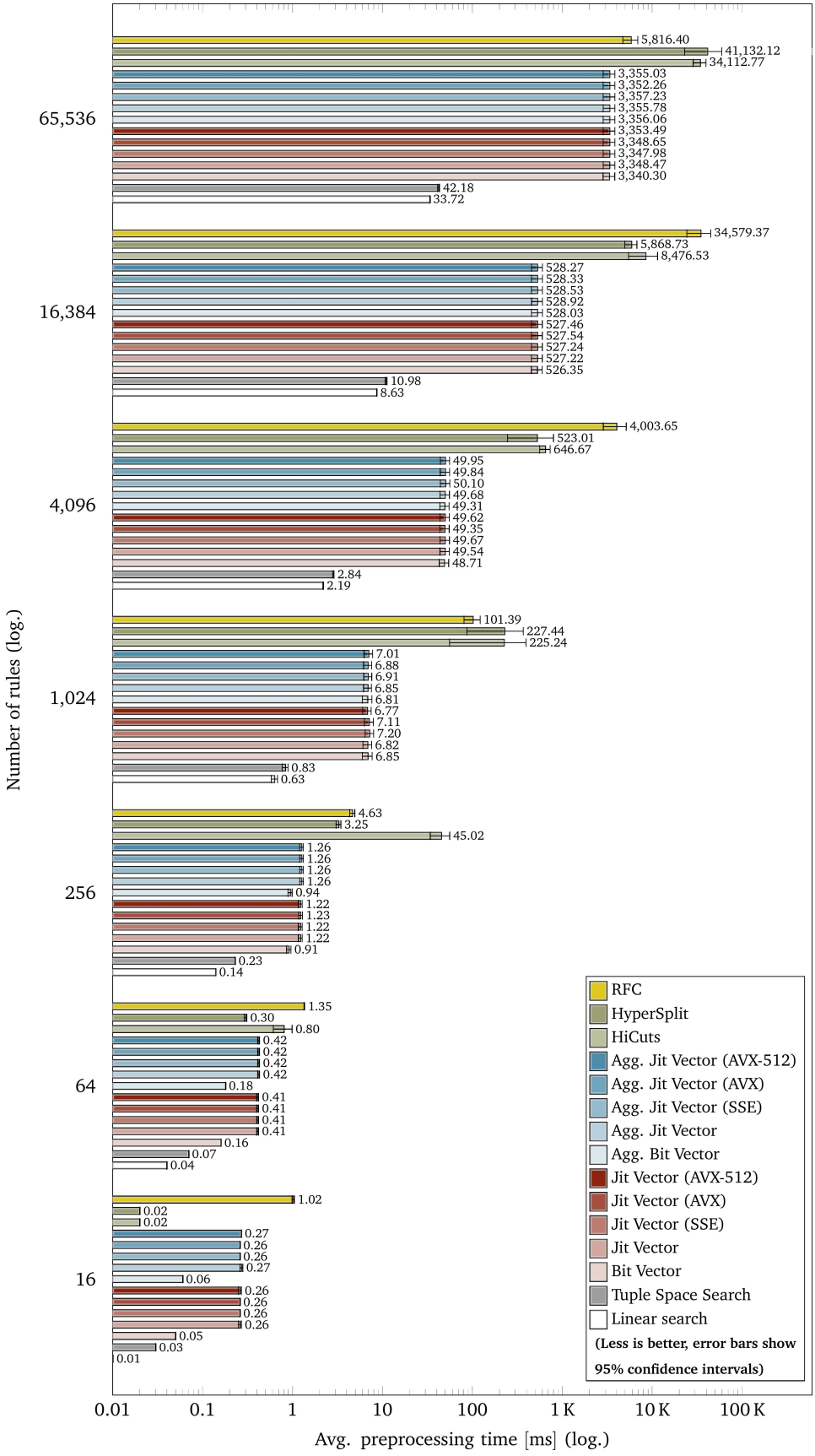


Fig. 5.7: Average preprocessing times of algorithm search data structures for different rule set sizes. Note that the RFC, HyperSplit, and HiCuts results only show results of successful search data structure builds.

5.5 Limitations

Generally, the proposed Jit Vector Search approach achieves better classification performance at the cost of larger memory footprints and higher preprocessing times, while still keeping its scalability, in contrast to decision tree algorithms [63, 107] or RFC [62]. Despite its performance gains, Jit Vector Search comes at high costs in terms of memory footprint and preprocessing time for smaller rule set sizes. While this is not a problem for scenarios where the rule set only changes seldom or moderately often, it might render the usage for our proposed approach impossible for highly dynamic environments, such as SDNs. In fact, we address this issue in Chapter 6.

Furthermore, dynamic code generation may lead to security concerns in certain applications, especially when the generated code is executed with kernel privileges. However, this seems to be a minor issue, especially when we take current development in the Linux kernel into account, which also uses dynamic code generators and specific static checkers to validate the generated instruction stream [34, 99]. In fact, it is always possible to ensure the validity of the generated binary search trees and lookup tables, as they never include any backward jumps and do not contain function calls.

Although Jit Vector Search is primarily designed to efficiently solve the Geometric Packet Classification Problem, it can be adjusted to also tackle the Complex Packet Classification Problem. This can be achieved through iteration over the result vector and executing potentially existing complex checks that belong to rules with set bits.

6 The SFL Classification Algorithm

Two of the most difficult challenges a classification system can face are the line speed packet processing requirement and the ability to quickly process rule set updates, especially when used in dynamic environments. Many existing approaches to packet classification mainly aim to to accelerate the classification process, ranging from fast classification algorithms [31, 62, 63, 76, 107, 128] and rule set optimization techniques [49, 12, 13, 65, 84, 88] to hardware-centric approaches [3, 56, 15, 136, 138]. Most of these works require significant preprocessing times to set up their search data structures, which in turn can be traversed quickly when a packet enters the classification system. In consequence, they provide excellent lookup performance in setups where the rule set does not change often, such as static security policies. However, if the classification system is used in dynamic environments with frequent rule set changes at run time, such as SDNs, the ability to quickly update the search structure is of paramount importance. Unfortunately, existing approaches that support dynamic updates either come with comparatively slow classification performance [61, 105, 127] or require specific hardware setups [2, 120, 136].

In this chapter, we contribute the *SFL* approach, which is a technique to equip a given classification algorithm with the ability to quickly process updates while still maintaining high lookup performance. Specifically, we can augment an arbitrary existing classification algorithm A (the *F*ast) with a list-based update buffer $\mathfrak{B}$ (the *L*azy), as sketched in Figure 6.1. Rule set updates for the classification system, which are applied at system run time, are not installed immediately in the search structure of A, but are inserted in the update buffer $\mathfrak{B}$ as well as in a *master rule set (the Small)*. When a network packet is to be classified, it is first matched using A's search data structure to compute a preliminary classification decision. Subsequently, this decision is checked based on the buffer and master rule set contents whether it is in conflict with a rule set update and is potentially modified. After sufficiently many updates have been collected, the classification data structure can be re-built once, thereby flushing the update buffer.

The main results of our evaluation are threefold: first, we demonstrate that existing fast classification algorithms fail to meet the requirements of highly dynamic environments, which results in severe throughput penalties. Second,

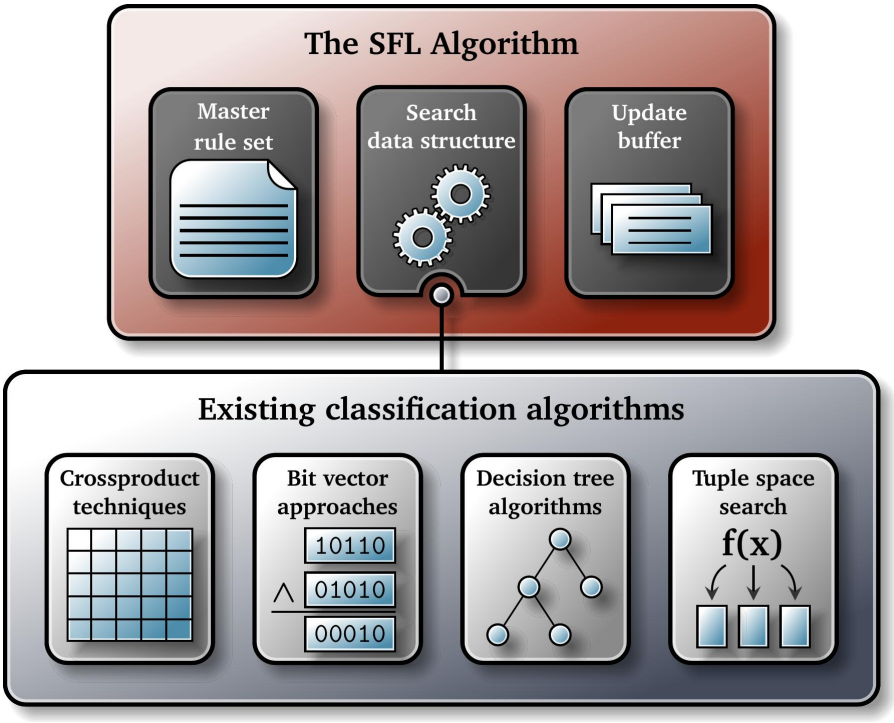


Fig. 6.1: Sketch of the SFL classification algorithm components.

we show that existing algorithms which support high update rates fall short in terms of throughput. Third, we show that fast SFL-"upgraded" algorithms perform significantly faster in dynamic environments than both existing fast and updateable classification algorithms. Specifically, some SFL-equipped algorithms can perform about an order of magnitude faster than the state-of-the-art dynamic algorithm *Tuple Space Search* [105, 127] while processing up to 60 updates per second.

6.1 System Interface

Before we dive into the details of the proposed SFL classification approach, we first describe a set of common basic procedures which most packet classification systems, such as firewalls or SDN switches, need to provide in order to be of practical use.

6.1.1 Fundamental Procedures

For a given classification system C and a classification algorithm A, which is run in C to process incoming network packets, we consider the following two procedures to be indispensable for practical operation:

First, it must be possible to *install a rule set* $\mathfrak{R}$ in the classification system C in order to guide C's classification process. As the classification process is driven by the algorithm A, a suitable *search data structure* $S_{A,\mathfrak{R}}$ must be computed upon rule set installation. We refer to the procedure, which translates a rule set $\mathfrak{R}$ to A's corresponding search data structure $S_{A,\mathfrak{R}}$ by $\mathit{spawn}(A, \mathfrak{R})$, with

$$S_{A,\mathfrak{R}} \leftarrow \mathit{spawn}(A, \mathfrak{R}). \tag{6.1}$$

When a network packet p is processed by C, p is used to query $S_{A,\mathfrak{R}}$ for the most highly prioritized matching rule. That is, the search data structure $S_{A,\mathfrak{R}}$ is used in the implementation of the *rule set decision function* $f_{\mathfrak{R}}$ (2.41), as described in Chapter 2. Although $S_{A,\mathfrak{R}}$ often only computes the index i^* of the most highly prioritized matching rule R_{i^*}, as described in Chapter 4, the corresponding action a^{i^*} can easily be retrieved from $\mathfrak{R}$ using the index i^* through, e. g., a direct table lookup. As such, we introduce the procedure $\mathit{classify}(p, S_{A,\mathfrak{R}})$, which computes the index i^* of the most highly prioritized matching rule R_{i^*} for a packet p by using A's search data structure, i. e.,

$$i^* \leftarrow \mathit{classify}(p, S_{A,\mathfrak{R}}). \tag{6.2}$$

6.1.2 Update Procedures

Having discussed the two fundamental *spawn* and *classify* procedures, which allow a classification system to be operated using an initial rule set

$$\mathfrak{R} = \langle R_1, \ldots, R_n \rangle, \tag{6.3}$$

we now consider *rule set updates*. More specifically, we focus on *rule insertions* and *rule deletions*, because these two operations allow for arbitrary rule set changes.

For a rule set $\mathfrak{R}$ with n rules and an index $i \in \{1, \ldots, n+1\}$, we model the insertion $\mathit{insert}(R', i, \mathfrak{R})$ of a new rule R' into $\mathfrak{R}$ as

$$\mathfrak{R}' \coloneqq \langle R_1, \ldots, R_{i-1}, R', R_i, \ldots, R_n \rangle \leftarrow \mathit{insert}(\mathfrak{R}, i, R'), \tag{6.4}$$

such that $|\mathfrak{R}'| = |\mathfrak{R}| + 1$.

Analogously to rule insertion, we use the following semantics for a rule deletion $\mathit{delete}(\mathfrak{R}, i)$, with $i \in \{1, \ldots, n\}$:

$$\mathfrak{R}' \coloneqq \langle R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n \rangle \leftarrow \mathit{delete}(\mathfrak{R}, i), \tag{6.5}$$

such that $|\mathfrak{R}'| = |\mathfrak{R}| - 1$.

An *update* to a rule set $\mathfrak{R}$ is either a deletion or an insertion. If the rule set $\mathfrak{R}$ changes to another rule set $\mathfrak{R}'$ due to an update operation Δ, $S_{A,\mathfrak{R}}$ must be adjusted by the classification system in order to correctly classify packets with respect to the rule set change. This can either happen incrementally or through a complete rebuild of the search data structure. Let the expression $\Delta(S_{A,\mathfrak{R}})$ denote the incrementally updated search structure, and $S_{A,\mathfrak{R}'}$ the search structure which is obtained through *spawn*$(A, \mathfrak{R}')$. It is indispensable that $\Delta(S_{A,\mathfrak{R}})$ and $S_{A,\mathfrak{R}'}$ are equivalent, i. e.,

$$\forall p \in P_{\text{MTU}} : \textit{classify}(p, \Delta(S_{A,\mathfrak{R}})) = \textit{classify}(p, S_{A,\mathfrak{R}'}). \tag{6.6}$$

6.1.3 Initial and Master Rule Set

We assume that a classification system C is always provided with an *initial rule set* $\mathfrak{R}_I$ that is installed in C using the *spawn* procedure. Also, we assume that C always has access to the currently active rule set which we call the *master rule set* $\mathfrak{R}_M$. The master rule set $\mathfrak{R}_M$ contains all rule set updates that have been executed since the installation of $\mathfrak{R}_I$. Furthermore, update operations, such as rule deletions or insertions, always target the master rule set, as it represents the current system configuration. That is, for the total sequence of update operations $\Delta_1, \Delta_2, \ldots, \Delta_k$ that have been issued since the installation of $\mathfrak{R}_I$, $\mathfrak{R}_M$ is defined as

$$\mathfrak{R}_M \coloneqq \Delta_k \left(\ldots \Delta_2(\Delta_1(\mathfrak{R}_I)) \right). \tag{6.7}$$

From an administrator's or SDN controller's point of view, $\mathfrak{R}_M$ reflects the desired behaviour of the classification system C. In this chapter, we assume that the classification system provides the following semantics: *as soon as the master rule set is changed either manually or programmatically, the classification system must process incoming packets with respect to the changed master rule set.* This property synchronizes the system's classification behaviour with the intended semantics of the installed rule set. Therefore, a non-hybrid system must propagate changes to $\mathfrak{R}_M$ immediately to the search data structure of the utilized classification algorithm.

Unfortunately, for many fast classification algorithms, both *spawn*$(A, \mathfrak{R}')$ as well as incremental updates take a considerable amount of time. As a consequence, the computation of $\Delta(S_{A,\mathfrak{R}})$ is as expensive as the costly execution of *spawn*$(A, \mathfrak{R}')$ in many cases. C, however, must respect the master rule set's semantics and

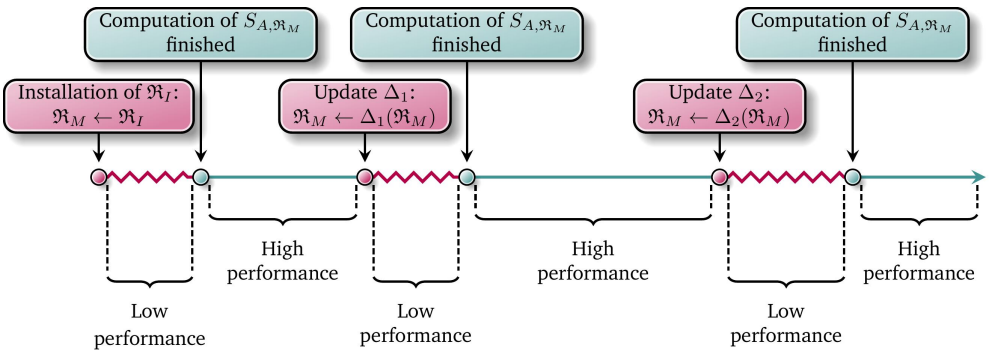


Fig. 6.2: Illustration of a classification system C's packet throughput over time when processing updates to the master rule set $\mathfrak{R}_M$. The red lines represent the durations in which C cannot utilize the search data structure $S_{A,\mathfrak{R}_M}$, as it is (re)computed. The green lines represent the durations in which C can fully benefit from the classification algorithm A's performance due to access to an up-to-date search data structure.

thus cannot rely solely on an outdated search data structure. As such, until the (possibly in a separate thread) recomputed search data structure is available, C must either stop processing packets or use the current master rule set as a fallback classifier, both of which can severely diminish the classification throughput in case of frequent rule set updates. This circumstance is sketched in Figure 6.2.

In the remainder of this chapter, we address this problem and propose a classification system design that allows, for a given *existing* classification algorithm A, the fast computation of $\Delta(S_{A,\mathfrak{R}})$ that is correct with respect to (6.6) *and* can be searched efficiently.

6.2 Update Buffer

In order to facilitate quick rule set updates, the SFL system relies on a data structure called the *update buffer* $\mathfrak{B}$. Instead of immediately computing the adjusted search data structure $S_{A,\mathfrak{R}'}$ in case of a rule set update Δ with

$$\mathfrak{R} \xrightarrow[\text{Update } \Delta]{} \mathfrak{R}', \tag{6.8}$$

SFL inserts the Δ operation into the update buffer $\mathfrak{B}$, which, in essence, is a linear list of *delayed updates*. We call each element in $\mathfrak{B}$ an *update node*, each of which is either an *insert node* or a *delete node*, depending on the type of the update. $\mathfrak{B}$ is structured in a way that allows to correct potential wrong classification results provided by the now outdated search data structure $S_{A,\mathfrak{R}}$, as we will explain in Section 6.3. In the remainder of this section, however, we focus on how rule insertions and deletions can be implemented within $\mathfrak{B}$.

6.2.1 Indices

In order to provide meaningful semantics for update nodes, we assign two indices to every rule R in the master rule set $\mathfrak{R}_M$, namely the *initial index* $\iota_I(R)$ and the *master index* $\iota_M(R)$. The initial index $\iota_I(R)$ refers to the position of the rule R in the initial rule set $\mathfrak{R}_I$, i. e.,

$$\iota_I(R) \coloneqq \begin{cases} R\text{'s position } (>0) \text{ in } \mathfrak{R}_I, \text{ if } R \in \mathfrak{R}_I \\ 0, \text{ otherwise} \end{cases} \tag{6.9}$$

The master index $\iota_M(R)$ is R's current position in the master rule set $\mathfrak{R}_M$. While $\iota_I(R)$ does not change until the *spawn* procedure is executed the next time, $\iota_M(R)$ can change due to rule insertions or deletions. In the remainder of this chapter, we use the implicit notation

$$R^{y(=\iota_I(R))}_{x(=\iota_M(R))} \tag{6.10}$$

to denote a rule R's initial and master indices.

For example, consider the initial rule set

$$\mathfrak{R}_M \leftarrow \mathfrak{R}_I = \langle R^1_1, R^2_2, R^3_3 \rangle. \tag{6.11}$$

If we insert the new rule R' at index 3, i. e., execute *insert($\mathfrak{R}_M$, 3, R')*, the master rule set changes to

$$\mathfrak{R}_M \leftarrow \langle R^1_1, R^2_2, R^0_3 (= R'), R^3_4 \rangle. \tag{6.12}$$

A subsequent deletion of the rule at master index 1 yields the following master rule set:

$$\mathfrak{R}_M \leftarrow \langle R^2_1, R^0_2 (= R'), R^3_3 \rangle \tag{6.13}$$

6.2.2 Rule Insertions

In order to insert a rule R at master index i, the SFL system first inserts R into the master rule $\mathfrak{R}_M$ set by executing *insert($\mathfrak{R}_M$, i, R)*. Next, the system creates an insertion node

$$N^{\text{insert}} = (\vec{R}, \psi(i)) \tag{6.14}$$

which stores a pointer $\vec{R}$ to rule R and a *reference index* $\psi(i)$. Descriptively, the reference index $\psi(i)$ is used to recognize all rules in $\mathfrak{R}_I$ which have a lower

```
1 function REFERENCE_INDEX(Master rule set 𝔑_M, Initial rule set 𝔑_I, Index i)
2     return min({j | R_k^j ∈ 𝔑_M ∧ i ≤ k ∧ j ≠ 0} ∪ {|𝔑_I| + 1})

3 function SFL_INSERT(Update buffer 𝔅, Master rule set 𝔑_M,
                      Initial rule set 𝔑_I, Index i, Rule R)
4     INSERT(𝔑_M, i, R_i^0)
5     ψ(i) ← REFERENCE_INDEX(𝔑_M, 𝔑_I, i)
6     ADD_INSERTION_NODE_TO_UPDATE_BUFFER(𝔅, N^insert = (R_i^0, ψ(i)))
```

Algorithm 6.1: Pseudocode for the SFL insertion procedure SFL_INSERT.

priority than the newly inserted rule R. More specifically, $\psi(i)$ is determined by

$$\psi(i) \coloneqq \min\Big(\underbrace{\{j \mid R_k^j \in \mathfrak{R}_M \ \wedge\ i \leq k \ \wedge\ j \neq 0\}}_{\substack{\text{The set of initial indices of rules in } \mathfrak{R}_I \\ \text{which are less highly prioritized than } R}} \cup \underbrace{\{|\mathfrak{R}_I| + 1\}}_{\text{Marker value}} \Big). \tag{6.15}$$

If $\psi(i) = |\mathfrak{R}_I| + 1$, then there is no rule in $\mathfrak{R}_I$ which is less highly prioritized than R. The reference index is of critical importance for the SFL classification process, because it is used as a termination criterion while iterating over the update buffer, which is described in Section 6.3.

In the last step, the SFL system inserts the newly created insertion node $N_k^{\text{insert}} = (\vec{R}, \psi(i))$ into the linear update buffer $\mathfrak{B}$, such that the following invariant is maintained:

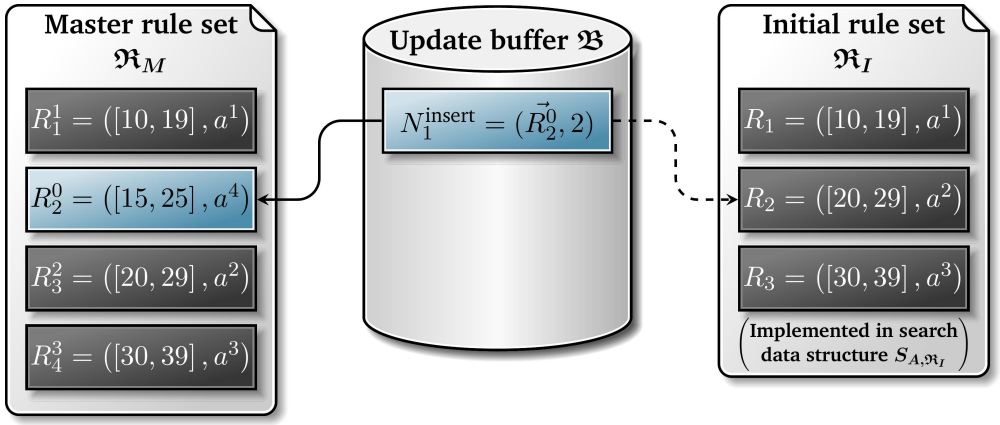
$$\forall\ N_i^{\text{insert}} = (\vec{R}_x, \cdot), N_j^{\text{insert}} = (\vec{R}_y, \cdot) \in \mathfrak{B} : x < y \iff \text{pos}(N_i^{\text{insert}}) < \text{pos}(N_j^{\text{insert}}), \tag{6.16}$$

with

$$pos(N_i) \tag{6.17}$$

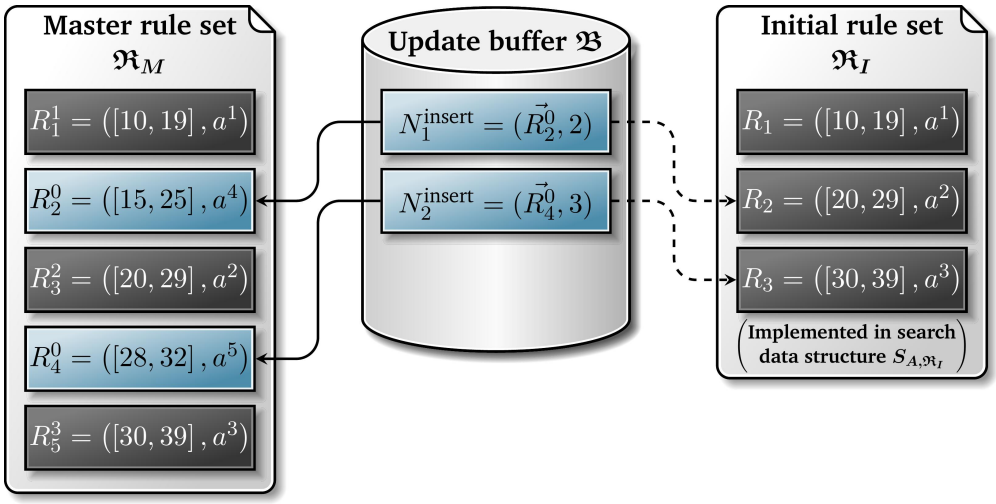
being an insertion node's position in the update buffer. In essence, Invariant 6.16 states that insertion nodes in $\mathfrak{B}$ have the same relative ordering as the corresponding rules in $\mathfrak{R}_M$, which is important for the SFL classification operation. The SFL rule insertion procedure is summarized in Algorithm 6.1. We illustrate the above described rule insertion process with a sequence of insertions in Figure 6.3.

(a) Situation immediately after the execution of *spawn*$(A, \mathfrak{R}_I)$. The update buffer $\mathfrak{B}$ is empty, and $\mathfrak{R}_M = \mathfrak{R}_I$. The search data structure $S_{A,\mathfrak{R}_I}$ implements the matching semantics of $\mathfrak{R}_I$.

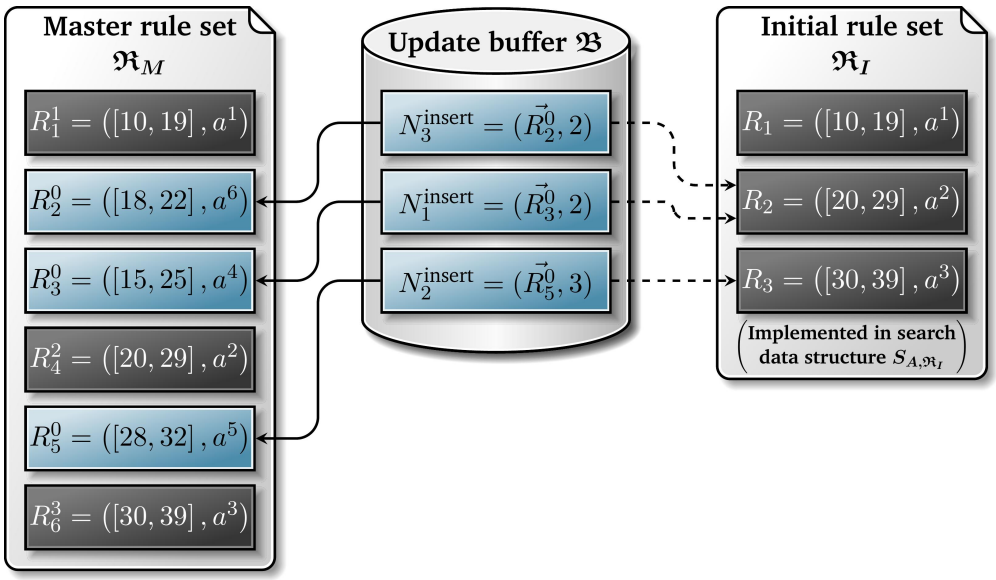


(b) Situation immediately after the execution of SFL_INSERT$(\mathfrak{B}, \mathfrak{R}_M, \mathfrak{R}_I, 2, R')$ with $R' = ([15, 25], a^4)$. $R' (= R_2^0)$ is added to $\mathfrak{R}_M$, and the update node $N_1^{\text{insert}} = (\vec{R}', 2)$ is inserted into $\mathfrak{B}$. Note that the master indices of less highly prioritized rules than R' are incremented.

Fig. 6.3: Illustration of the SFL rule insertion process, using a one-dimensional geometric rule set. The dark rules represent rules from the initial rule set $\mathfrak{R}_I$, while blue rules are dynamically added to the master rule set $\mathfrak{R}_M$ after *spawn*$(A, \mathfrak{R}_I)$. The solid lines represent the pointers stored in update nodes, while the dashed lines represent the update nodes' reference indices.



(c) Situation immediately after the execution of SFL_INSERT$(\mathfrak{B}, \mathfrak{R}_M, \mathfrak{R}_I, 4, R'')$ with $R'' = ([28, 32], a^5)$. $R''(= R_4^0)$ is added to $\mathfrak{R}_M$, and the update node $N_2^{\text{insert}} = (\vec{R}'', 3)$ is inserted into $\mathfrak{B}$. Note that the master indices of less highly prioritized rules than R'' are incremented.



(d) Situation immediately after the execution of SFL_INSERT$(\mathfrak{B}, \mathfrak{R}_M, \mathfrak{R}_I, 2, R''')$ with $R''' = ([18, 22], a^6)$. $R'''(= R_2^0)$ is added to $\mathfrak{R}_M$, and the update node $N_3^{\text{insert}} = (\vec{R}''', 2)$ is inserted into $\mathfrak{B}$. Note that the master indices of less highly prioritized rules than R'' are incremented. Furthermore, note that N_3 is inserted in front of N_1 in $\mathfrak{B}$, since R_2^0 is more highly prioritized than R_3^0.

Fig. 6.3: Illustration of the SFL rule insertion process, using a one-dimensional geometric rule set (continued).

6.2.3 Rule Deletions

We now turn our attention to the deletion of rules in the SFL context. In contrast to rule insertions, which *always* insert an update node into the update buffer $\mathfrak{B}$, deletions must distinguish between two different kinds of rules: *initial rules* and *dynamic rules*. For a rule $R_x^y \in \mathfrak{R}_M$, we say that R_x^y is an *initial rule* iff $y \neq 0$, i. e., if $R_y \in \mathfrak{R}_I$. Otherwise, if $y = 0$, i. e., if $R_x^y \in \mathfrak{R}_M \setminus \mathfrak{R}_I$, then R_x^y has been added to the system *after* the execution of *spawn*$(A, \mathfrak{R}_I)$ and is thus called a *dynamic rule*.

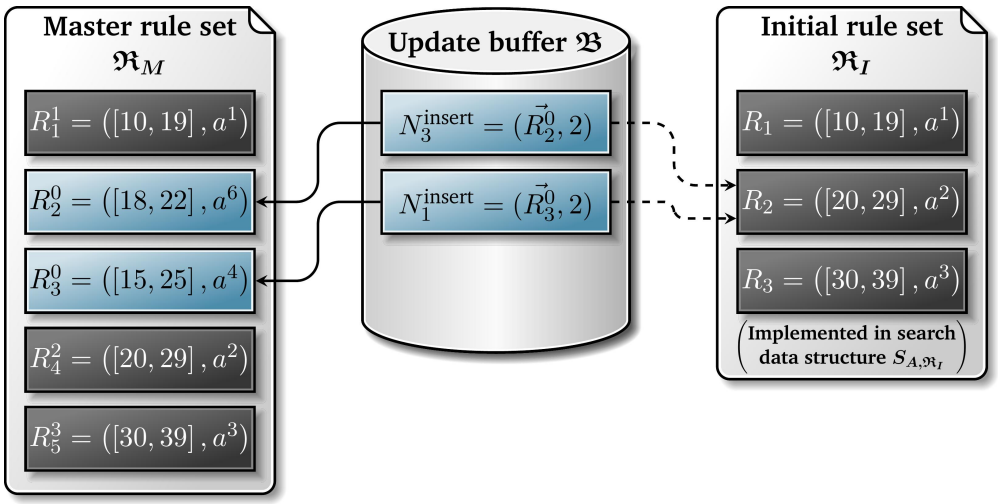
When a rule R_x^y with master index x is to be deleted from the master rule set using *delete*$(\mathfrak{R}_M, x)$, the SFL system first determines whether R_x^y is dynamic. If this is the case, then there must be a corresponding insertion node $(\vec{R_x^y}, \cdot)$ in the update buffer $\mathfrak{B}$, which was previously added by the SFL system in order to correct potentially wrong classification results by the search data structure $S_{A,\mathfrak{R}_I}$. Accordingly, the removal of $(\vec{R_x^y}, \cdot)$ from $\mathfrak{B}$ as well as the deletion of R_x^y from $\mathfrak{R}_M$ via the execution of *delete*$(\mathfrak{R}_M, x)$ are sufficient to implement the deletion of dynamic rules, because it entirely reverses the change in matching semantics through the previous insertion of R_x^y.

However, if R_x^y is an initial rule, the SFL system needs to add information to the update buffer in order to cope with potentially incorrect classification results from $S_{A,\mathfrak{R}_I}$. For example, consider the situation that $S_{A,\mathfrak{R}_I}$ computes the matching index $i^* = y$ after the deletion of R_x^y, which is clearly incorrect (remember that the rule $R_x^y \in \mathfrak{R}_M$ corresponds to the rule $R_y \in \mathfrak{R}_I$). Therefore, the system inserts a deletion node $N^{\text{delete}} = y$ into $\mathfrak{B}$, which stores the initial index of the deleted rule as its reference index. Similar to insertion nodes, we assert the following invariant for the position of the deletion node inside of $\mathfrak{B}$:

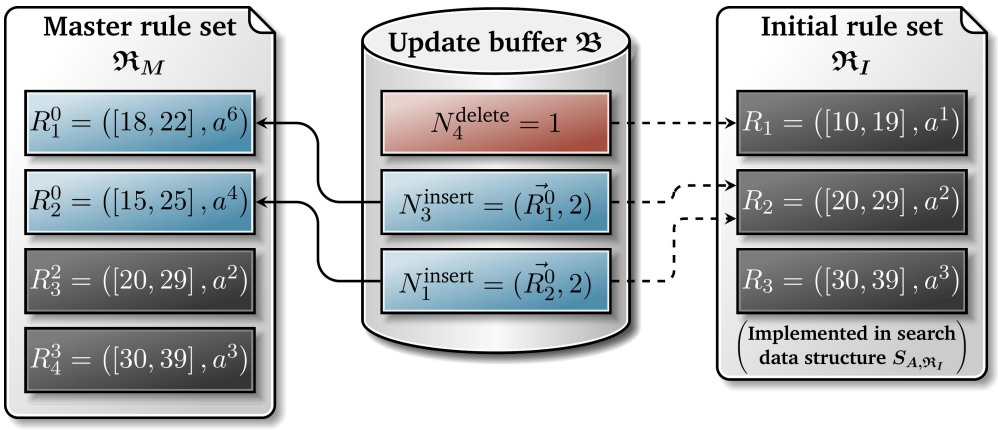
$$\begin{aligned} \forall\, N_i^{\text{delete}} = x, N_j^{\text{delete}} = y \in \mathfrak{B} : x < y &\iff \text{pos}(N_i^{\text{delete}}) < \text{pos}(N_j^{\text{delete}}) \\ \forall\, N_i^{\text{delete}} = x, N_j^{\text{insert}} = (\cdot, y) \in \mathfrak{B} : x \leq y &\iff \text{pos}(N_i^{\text{delete}}) < \text{pos}(N_j^{\text{insert}}) \end{aligned} \tag{6.18}$$

```
1 function SFL_DELETE(Update buffer 𝔅, Master rule set ℜ_M, Index i)
2     if ∃ R_i^0 ∈ ℜ_M then
3         REMOVE_FROM_UPDATE_BUFFER(𝔅, N^insert = (R⃗_i^0, ·))
4     else
5         ADD_DELETION_NODE_TO_UPDATE_BUFFER(𝔅, N^delete = i)
6     DELETE(ℜ_M, i)
```

Algorithm 6.2: Pseudocode for the SFL deletion procedure SFL_DELETE.



(a) Situation immediately after the execution of SFL_DELETE$(\mathfrak{B}, \mathfrak{R}_M, 5)$, which removed the update node N_2 from $\mathfrak{B}$ and R_5^0 from $\mathfrak{R}_M$. Note that the master indices of less highly prioritized rules than R_5^0 are decremented.



(b) Situation immediately after the execution of SFL_DELETE$(\mathfrak{B}, \mathfrak{R}_M, 1)$, which removed R_1^1 from $\mathfrak{R}_M$ and inserted the update node $N_4^{\text{delete}} = 1$ into $\mathfrak{B}$. Note that the master indices of less highly prioritized rules than R_1^1 are decremented.

Fig. 6.4: Illustration of the SFL rule deletion process, continued from Figure 6.3. The dark rules represent rules from the initial rule set $\mathfrak{R}_I$, blue rules are dynamically added to the master rule set $\mathfrak{R}_M$ after *spawn*$(A, \mathfrak{R}_I)$. The solid lines represent the pointers stored in update nodes, while the dashed lines represent the update nodes' reference indices.

Descriptively, Invariant 6.18 first states that deletion nodes are sorted with respect to their reference indices. Second, deletion nodes and insertion nodes are also sorted with respect to their reference indices, but deletion nodes are always placed in front of insertion nodes in case of reference index equality. We will revisit these invariants during the description of the SFL classification process in Section 6.3.

The pseudocode for the above described deletion procedure is described in Algorithm 6.2. Moreover, we continue our running example from Figure 6.3 in Figure 6.4, which illustrates the two different deletion cases.

6.3 Classification Process

Having introduced the structure of the SFL update buffer in the previous section, we now turn our attention to the SFL classification process. In order to classify an incoming packet p, the SFL system first uses algorithm A's search data structure $S_{A,\mathfrak{R}_I}$ to compute a preliminary matching index i^{tmp}. However, due to the fact that since the computation of $S_{A,\mathfrak{R}_I}$, i. e., the last execution of *spawn*$(A, \mathfrak{R}_I)$, several updates may have been installed in the update buffer $\mathfrak{B}$ due to changes to the master rule set $\mathfrak{R}_M$, i^{tmp} may no longer be correct with respect to $\mathfrak{R}_M$. For instance, when considering the situation sketched in Figure 6.4b, a packet p with the one-dimensional header $h^p = 15$ matches on rule $R_1 \in \mathfrak{R}_I$, which is not correct with respect to $\mathfrak{R}_M$ due to the previous deletion of R_1. Instead, a look into the master rule set reveals that the correct matching index would be $i^* = 2$.

Therefore, the SFL system needs to potentially correct the preliminary matching index by using the update information stored in the update buffer in a second step. To this end, the update nodes in $\mathfrak{B}$ are linearly traversed in ascending order with respect to the nodes' reference indices to determine if one of the following situation occurs:

1. The packet p matches on a dynamic rule $R_x^0 \in \mathfrak{R}_M$, such that R_x^0 is more highly prioritized than $R_y^{i^{\text{tmp}}}$.

2. The rule $R_y^{i^{\text{tmp}}}$ has been deleted and therefore does not exist in the master rule set.

3. Neither (1) nor (2) occurs, but the matching index computed by $S_{A,\mathfrak{R}_I}$ potentially needs to be corrected because $R_{i^{\text{tmp}}}$'s representation in the master rule set $R_y^{i^{\text{tmp}}}$ may now be stored at a different index $y \neq i^{\text{tmp}}$.

For the remainder of this section, we use the term $\Psi(N)$ to refer to an update node's reference index, with

$$\Psi(N) := \begin{cases} \psi, \text{ if } N \text{ is an insertion node with } N = (\cdot, \psi), \\ y, \text{ if } N \text{ is a deletion node with } N = y. \end{cases} \tag{6.19}$$

Situation 1. If the update buffer traversal yields an insertion node $N = (\vec{R}_x^0, \psi)$ with $\psi \leq i^{\text{tmp}}$, we must test whether rule R_x^0 matches p, because R_x^0 is more highly prioritized than $R_y^{i^{\text{tmp}}}$. If R_x^0 indeed matches p, we know that $i^* = x$ and that we can terminate the classification process.

Situation 2. If the update buffer traversal yields a deletion node $N_i = i^{\text{tmp}}$, we know that the rule $R_{i^{\text{tmp}}} \in \mathfrak{R}_I$, has been deleted. Therefore, we must locate the most highly prioritized rule R_{i^*} in the master rule set $\mathfrak{R}_M$. As such, the SFL approach applies a linear search over the master rule set $\mathfrak{R}_M$ in this case, because the linear search can directly operate on $\mathfrak{R}_M$ itself without requiring a sophisticated search data structure. Furthermore, the update buffer structure allows us to potentially prune the expensive linear search if one of the following conditions is met:

I If there is an insertion node $N_j \in \mathfrak{B} = (\vec{R}_x^0, y)$ with $\text{pos}(N_j) = \text{pos}(N_i) + 1$, we know that the rule R_x^0 is the next potential matching candidate for p due to the node ordering invariants (6.16) and (6.18). Therefore, we can safely begin the linear search at index x in the master rule set.

II If (I) is not met, let $N_j = (\vec{R}_x^0, y)$ be the last traversed insertion node before N_i. If such a node exists, we can safely begin the linear search at index $x + 1$ in the master rule set, because we already know that the rule R_x^0 does not match p (see Situation 1).

Only if neither (I) nor (II) holds, we execute a full linear search over the master rule set, as shown in Algorithm 6.3.

Situation 3. Because the nodes in $\mathfrak{B}$ are ordered with respect to Ψ, the traversal of $\mathfrak{B}$ can be stopped if an update node N with

$$\Psi(N) > i^{\text{tmp}} \tag{6.20}$$

is encountered. Remember that an insertion node N's reference index points to the most highly prioritized rule in $\mathfrak{R}_I$, in front of which N's dynamic rule is inserted. Furthermore, a deletion node's reference index points to a deleted rule in

$\mathfrak{R}_I$. If Condition (6.20) holds, then the rule set change encoded in N has no effect on the currently classified packet p, as p matches on a more highly prioritized rule in $\mathfrak{R}_I$. Therefore, due to the node ordering invariants (6.16) and (6.18), we can safely terminate the update buffer traversal in this situation. Moreover, we can compute p's final matching index i^* with

$$i^* = i^{\text{tmp}} + \alpha, \tag{6.21}$$

with

$$\begin{aligned} \alpha \coloneqq &\text{ (number of traversed insertion nodes)} \\ &- \text{ (number of traversed deletion nodes).} \end{aligned} \tag{6.22}$$

as neither situation (1) nor (2) has occurred until now. This is explained by the fact that every visited insertion node up to this point corresponds to a more highly prioritized rule than $R_{i^{\text{tmp}}} \in \mathfrak{R}_I$, which are placed in front of $R'_{i^{\text{tmp}}}s$ representation in the master rule set. Analogously, visited deletion nodes represent deletions of more highly prioritized rules in $\mathfrak{R}_I$, which decrement the position of $R_{i^{\text{tmp}}}$'s representation in the master rule set. Of course, the same reasoning can be applied if the entire update buffer has been traversed.

We conclude this section with an illustration of the SFL classification procedure pseudocode, which is shown in Algorithm 6.4, as well as two example classifications, using the situation sketched in Figure 6.4b.

Example 1 ($h^p = 18$). As *classify*$(S_{A,\mathfrak{R}_I}, p)$ yields the index $i^{\text{tmp}} = 1$ and because $N_1 = 1$, the master rule set is searched linearly, resulting in the matching index

$$i^* = 1, \tag{6.23}$$

which is correct with respect to $\mathfrak{R}_M$.

Example 2 ($h^p = 26$). *classify*$(S_{A,\mathfrak{R}_I}, p)$ yields the index $i^{\text{tmp}} = 2$. Since the initial rule R_2 has not been deleted, and because no dynamically inserted rule matches, the entire update buffer is traversed, leading to the index correction $\alpha = -1 + 1 + 1 = 1$. Therefore, the matching index is determined as

$$i^* = i^{\text{tmp}} + \alpha = 2 + 1 = 3, \tag{6.24}$$

which is correct with respect to $\mathfrak{R}_M$.

```
1  function RESOLVE_DELETE_MATCH(Update buffer 𝔅, Master rule set ℜ_M,
                                  Update node N_i = y, Packet p)
2      // First, test whether linear search can begin from a subsequent dynamic rule (I).
3      if ∃ N_j ∈ 𝔅 ∧ pos(N_j) = pos(N_i) + 1 ∧ N_j = (R⃗^0_x, y) then
4          i^start ← x
5      // Second, check whether linear search can begin from the successor rule
6      // to a more highly prioritized dynamic rule (II).
7      else if ∃ N_j^insert ∈ 𝔅 ∧ pos(N_j) < pos(N_i) then
8          l ← max({k | N_k^insert = (R⃗^0_{x_k}, ·) ∈ 𝔅 ∧ pos(N_j) < pos(N_i)})
9          i^start ← x_l + 1
10     // If neither Condition (I) nor (II) is met, we start the linear search at the first rule.
11     else
12         i^start ← 1
13     return LINEAR_SEARCH(ℜ_M, p, begin search at index i^start)
```

Algorithm 6.3: Pseudocode for the procedure RESOLVE_DELETE_MATCH.

```
1  function SFL_CLASSIFY(Update buffer 𝔅, Master rule set ℜ_M,
                          Search data structure S_{A,ℜ_I}, Packet p)
2      // First, we compute matching index i^tmp using the search data structure S_{A,ℜ_I}.
3      i^tmp ← CLASSIFY(S_{A,ℜ_I}, p)
4      // Next, we iterate over 𝔅 in order to correct the potentially outdated index i^tmp.
5      α ← 0
6      for j = 1 to |𝔅| do
7          N_i ← RETRIEVE_NODE_FROM_POSITION(𝔅, j)
8          // We test whether the update node N_i can change the classification result.
9          // If this is not the case, we stop traversing 𝔅.
10         if Ψ(N_i) > i^tmp then
11             break
12         // If N_i is an insertion node, we test whether the inserted rule R^0_x matches p.
13         // If this is true, we return the master index x, else we increment the offset α.
14         if N_i = (R⃗^0_x, ·) then
15             if R^0_x matches p then
16                 return x
17             else
18                 α ← α + 1
19         // If N_i is a deletion node, we test whether the rule R_{i^tmp} ∈ ℜ_I was deleted.
20         // If this is true, we traverse the master rule set ℜ_M to classify p, as shown
21         // in Algorithm 6.3.
22         // Otherwise, we decrement the offset α.
23         if N_i = y then
24             if y = i^tmp then
25                 return RESOLVE_DELETE_MATCH(𝔅, ℜ_M, N_i, p)
26             else
27                 α ← α - 1
28     // Finally, we return the by the offset α adjusted matching index.
29     return i^tmp + α
```

Algorithm 6.4: Pseudocode for the SFL classification procedure SFL_CLASSIFY.

6.4 Forcing

The downside of storing rule set updates in the update buffer $\mathfrak{B}$ is the degradation of classification performance with an increasing number of updates. Each update except for the deletion of dynamically added rules causes the creation of a new update node which may be traversed during the lookup process. After several updates, the performance penalty induced by the growing size of $\mathfrak{B}$ requires a rebuild of the utilized search data structure in order to restore matching performance to the level usually reached by the classification algorithm A. A rebuild is executed by computing the search data structure $S_{A,\mathfrak{R}_M}$ that encodes the semantics of the current master rule set $\mathfrak{R}_M$ using *spawn*$(A, \mathfrak{R}_M)$ Furthermore, we issue the assignment

$$\mathfrak{R}_I \leftarrow \mathfrak{R}_M, \tag{6.25}$$

that is, the *new* initial rule set $\mathfrak{R}_I$ is identical to $\mathfrak{R}_M$ This process, which we call *forcing (the update buffer)*, is sketched in Figure 6.5.

However, depending on A, forcing can take a considerable amount of time due to the execution of *spawn*$(A, \mathfrak{R}_M)$. Therefore, these rebuilds should not be triggered too often. On the other hand, forcing too seldom may result in poor classification performance over time because of the additional lookup work induced by the growing update buffer. Thus, we propose to use an *update threshold* δ, which determines how many update operations can be applied to the update buffer before the search data structure of A is rebuilt. Note that if $\delta = 0$, then the SFL approach behaves exactly like a non-SFL classification system.

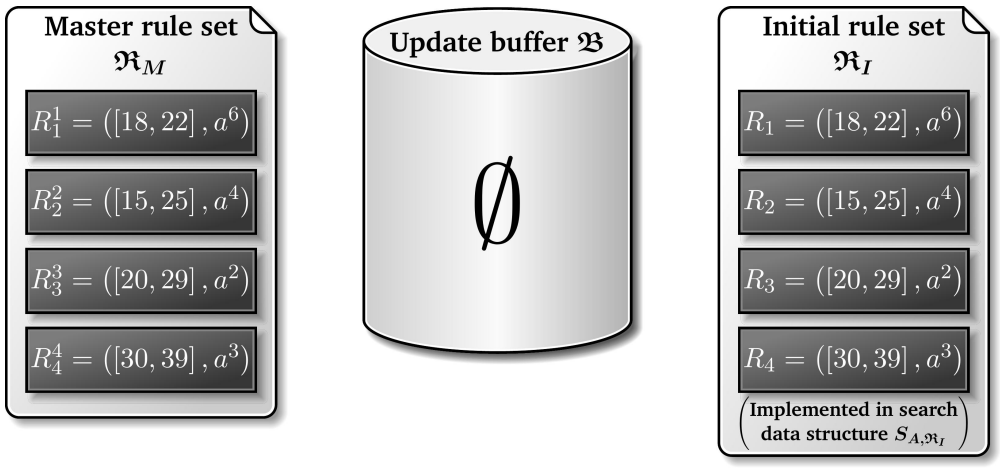


Fig. 6.5: Master rule set, initial rule set, and update buffer after forcing the update buffer from Figure 6.4.

6.5 Performance Characteristics

In this section, we review the main performance characteristics of the SFL algorithm and compare them to existing related work. For the remainder of this section, let the term

$$\mathit{SFL}(A) \tag{6.26}$$

denote an SFL system that wraps the classification algorithm A. The SFL(A) classification algorithm can perform δ consecutive rule set updates in $\mathcal{O}(\delta + n)$ steps per update, as both the update buffer $\mathfrak{B}$ and the master rule set $\mathfrak{R}_M$ are modified. Here, n is the number of rules in $\mathfrak{R}_M$, while δ is the update threshold of $\mathfrak{B}$. As the $(\delta+1)$-st update operation requires a rebuild of the search data structure of the algorithm A, it is as expensive as executing $\mathit{spawn}(A, \mathfrak{R}_M)$. SFL($A$)'s classification performance depends on the time needed by $\mathit{classify}(p, S_{A,\mathfrak{R}_I})$, since A's search structure is queried first. Subsequently, the update buffer $\mathcal{B}$ is searched, which takes at most δ steps if no delete node is hit, and otherwise at most n steps due to the subsequent linear search in the master rule set $\mathfrak{R}_M$. Note, however, that this linear search starts may not necessarily have to start at the first rule in $\mathfrak{R}_M$, which can reduce the cost of the linear search. Hence, classifying a packet p using SFL(A) takes $\mathit{classify}(p, S_{A,\mathfrak{R}_I})$ plus $\mathcal{O}(\delta + n)$ steps in the worst case. In terms of memory requirements, the SFL approach adds comparatively little overhead to existing techniques in form of the update nodes stored in $\mathfrak{B}$. Table 6.1 summarizes the runtime complexities for classification and update operations.

6.6 Evaluation

In this section, we evaluate the proposed SFL approach by comparing it to the existing classification algorithms Linear Search, Tuple Space Search [127], Bit Vector Search [76], HyperSplit [107], and RFC [62] in terms of throughput and update responsiveness. To this end, we conduct a series of experiments in order to evaluate the classification and update performances using our implementations of each of the abovementioned algorithms as well as their SFL-enhanced variants in a system simulation.

6.6.1 Experiment Setup

In order to conduct our experiments, we use a self-implemented classification system simulator that is capable of simultaneously executing packet classifications, issuing rule set updates, processing rule set updates, and counting the number of processed packets and updates in a multithreaded userspace tool. The system

Approach	Classification operation	Data structure creation	Data structure update
		Related work	
Linear Search	$\mathcal{O}(d \cdot n)$	$\mathcal{O}(d \cdot n)$	$\mathcal{O}(n)$
Tuple Space Search [127]	$\mathcal{O}(m)$ (amortized)	$\mathcal{O}(d \cdot n)$	$\mathcal{O}(1)$ (amortized)
Bit Vector Search [76]	$\mathcal{O}\left(d \cdot \log(n) + d\left\lceil \frac{n}{w} \right\rceil\right)$	$\mathcal{O}(d \cdot n^2)$	$\mathcal{O}(d \cdot n^2)$
HyperSplit [107]	$\mathcal{O}(d \cdot \log(2n+1))$ (assumed)	$\mathcal{O}(n^d)$ (assumed)	$\mathcal{O}(n^d)$ (assumed)
RFC [62]	$\mathcal{O}(d)$	$\mathcal{O}(n^d)$	$\mathcal{O}(n^d)$
		Proposed approach	
SFL(A)	$\mathit{classify}_A + \mathcal{O}(\delta + n)$	spawn_A	**($\leq \delta$ ops)** $\mathcal{O}(\delta + n)$ **(($\delta+1$)st op)** spawn_A

n: number of rules d: number of dimensions w: machine word width m: number of tuples
$\mathit{classify}_A$: classification time with algorithm A spawn_A: search data structure creation for algorithm A

Tab. 6.1: SFL performance characteristics, in comparison to related work.

incorporates the abovementioned algorithms as well as their SFL variants. For a specified classification algorithm A, an initial rule set $\mathfrak{R}_I$, a header trace T, a sequence of update operations $\Delta_1, \Delta_2, \ldots, \Delta_k$, an update buffer capacity δ, and a duration D, the system loops over the header fields in T for the duration D and classifies them. After each time period of D/k, the ith update operation Δ_i is issued to the system's update queue, which is constantly polled by a dedicated thread in order to conduct the issued update operations. Each polled update operation is immediately applied to the master rule set $\mathfrak{R}_M$, which is also stored within the system. If a non-SFL classification algorithm is used, every update operation triggers a search data structure rebuild. On the other hand, if an SFL-enhanced algorithm is evaluated, the system implements the update mechanics as described in Section 6.4. During a search data structure rebuild, which are also executed in a separate thread, or when the SFL update buffer is forced, the system uses the master rule set as a slow fallback classifier.

We use the tool *ClassBench* [132] in order to generate the rule sets and traces used in our experiments. All rule sets are five-dimensional and specify source and destination IPv4 subnets, the layer four protocol field, as well as source and destination port ranges. Every generated rule set contains $2000 + k$ rules, where k is the number of dynamic rule insertions. For every rule set size, we generate ten different rule sets. The initial rule sets always contain 2000 randomly chosen

rules from the generated rule sets, while the remaining rules are inserted during the simulation. We simulate rule deletions by randomly deleting from the initial rule set. For each rule set, we also generate a corresponding traces with 100K headers that are uniformly distributed over the rules in the rule set.

All experiments are conducted on a machine with an Intel Xeon E5-1660v3 CPU with eight physical cores and 128 GB of main memory, running Ubuntu 14.04 LTS. The system simulator and the evaluated algorithms are implemented in C, the code is compiled using `gcc 4.8.4` using the flags

```
-Wall -Werror -pedantic-errors -std=gnu99 -O3 -pthread.
```
(6.27)

The system simulator reads its entire configuration from text files, including the rule sets and header traces. As such, neither network access nor real packet I/O is required to run the simulation.

6.6.2 Throughput and Update Responsiveness

In our first experiment, we study the classification throughput and the update responsiveness of the SFL approach and the other algorithms in a system simulation over a time period of ten seconds. To this end, we either apply either 10, 100, 300, or 600 rule insertions or deletions to the installed rule set in fixed intervals, while the classification system is constantly fed with packet headers as fast as possible by looping over the trace. Finally, we measure the number of classified headers as well as the number of successfully executed updates during system runtime. In this experiment, the δ parameter was set to 20.

Figure 6.6 and Figure 6.7 show the average number of executed insertion and deletion operations per second, respectively. It can be seen that, on the one hand, the dynamically updateable Linear Search and Tuple Space Search algorithms are able to process almost every issued update. On the other hand, the non-dynamically updateable Bit Vector Search, RFC, and HyperSplit algorithms hardly meet the required number of issued update operations because of their costly data structure rebuilds. In contrast, their SFL counterparts provide a significantly higher update rate, due to the fact that only every 21st update forces the update buffer and results in a data structure rebuild. All remaining update operations can be executed quickly by simply modifying the master rule set and the update buffer. For instance, the number of updates processed by SFL(RFC) is 4.5$\times$ higher in the rule insertion benchmark and 3.9$\times$ higher in the deletion case, when 60 updates per second are issued. The deletion factor is smaller in the deletion case because the size of the master rule set shrinks during the experiment, which leads to faster

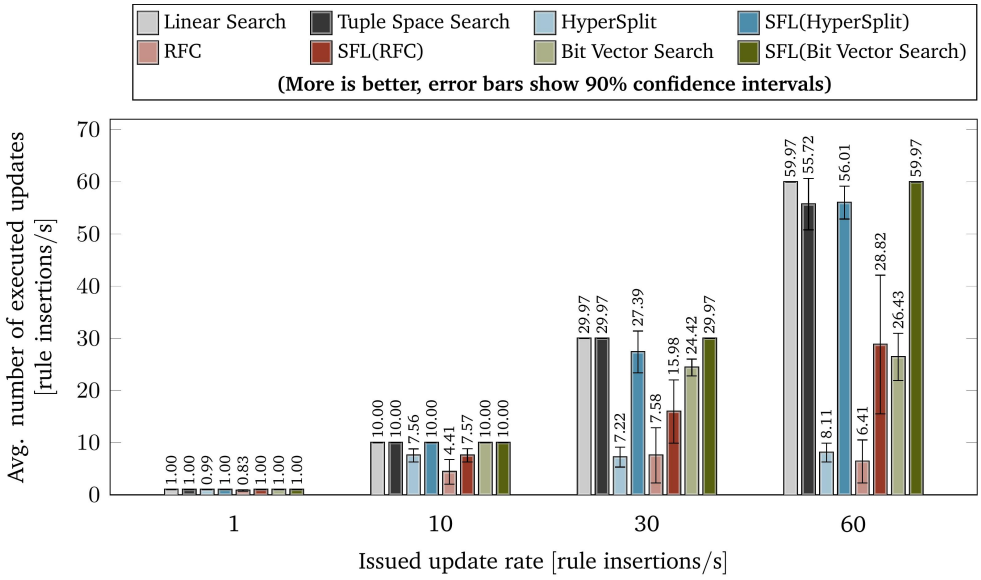


Fig. 6.6: Average number of insertion operations executed by the system simulator for different SFL and non-SFL algorithms.

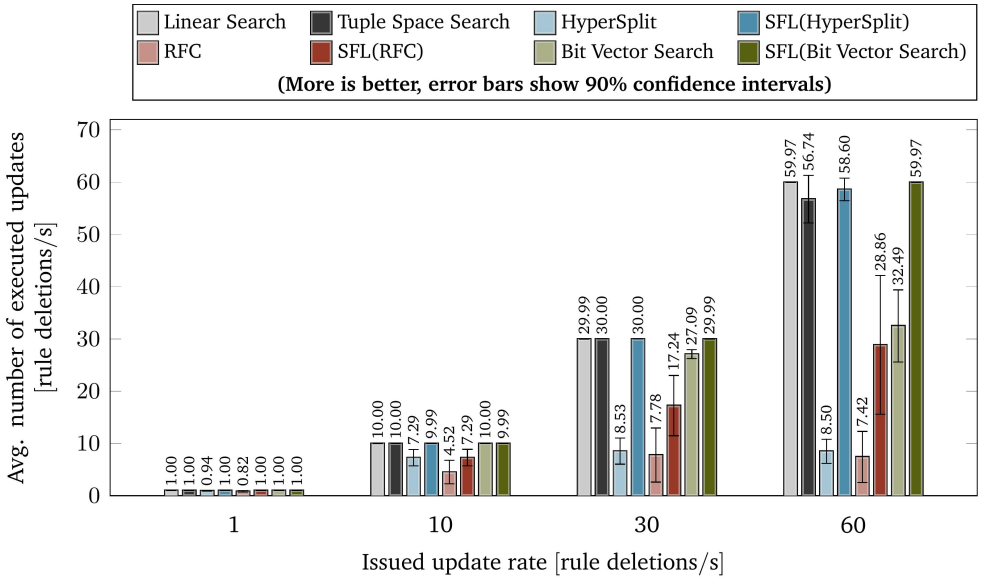


Fig. 6.7: Average number of deletion operations executed by the system simulator for different SFL and non-SFL algorithms.

RFC data structure rebuilds. Generally, we see that for all issued update rates, the SFL-enhanced algorithms either match or, most often, significantly outperform their classic counterparts.

Next, we examine the achievable classification throughput, which is illustrated in Figure 6.8 and Figure 6.9 for the insertion and deletion case, respectively. The figures clearly reveal the negative performance impact of frequent update operations on the classification performance of classification algorithms without support for dynamic updates: with an increasing number of issued updates, the classifica-

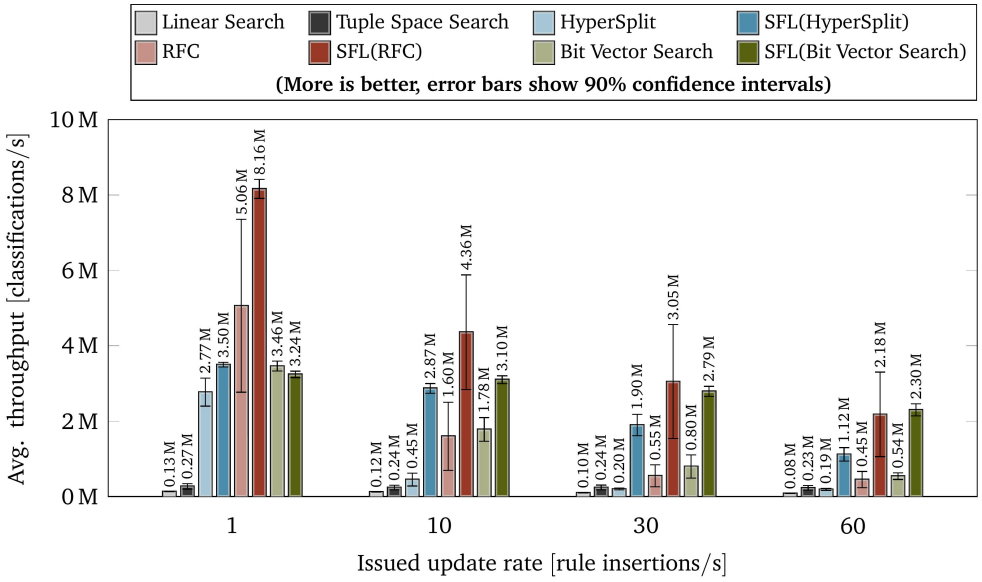


Fig. 6.8: Average classification throughput of the system simulator for different SFL and non-SFL algorithms for rule insertions.

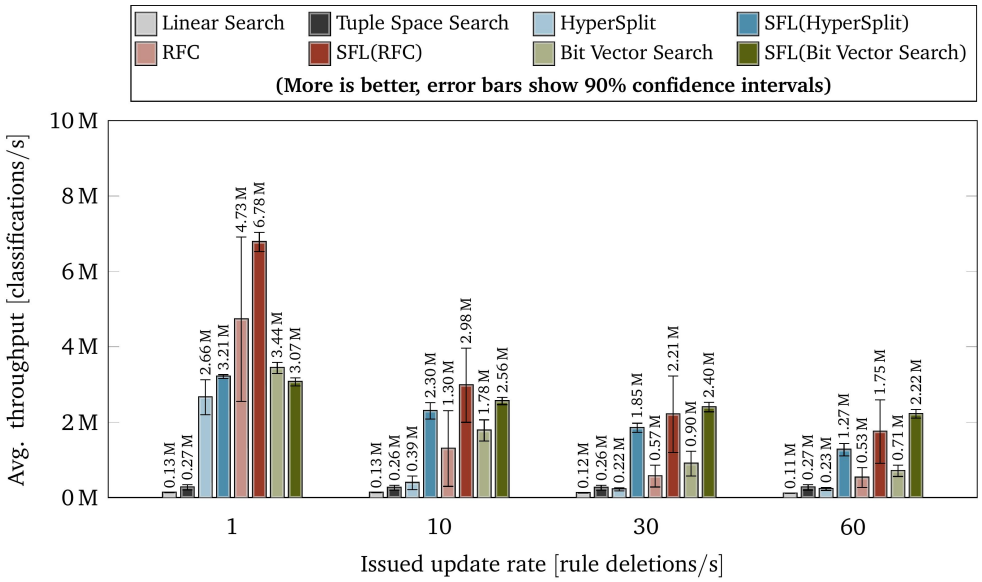


Fig. 6.9: Average classification throughput of the system simulator for different SFL and non-SFL algorithms for rule deletions.

tion throughput of Bit Vector Search, RFC, and HyperSplit significantly drops, because their search data structures are recomputed increasingly often. This, in turn, forces the system simulator to fall back to linearly searching the master rule set during the recomputations, which explains the classification performance penalties. We also observe that, while the SFL-enhanced algorithms also suffer from decreased classification performance with an increasing number of updates, their throughputs are clearly superior to those of their classic counterparts. Again, this is explained by the comparatively few data structure rebuilds that must be executed during system runtime.

6.6.3 Influence of the δ parameter

In our second experiment, we examine how the choice of δ influences the achievable classification throughput. To this end, we used the same experimental setup as in Section 6.6.2, but vary the δ parameter from 0 to 100 in steps of 10. Also, we fix the total number of issued rule insertions/deletions to 100 per evaluation run. Figure 6.10 and Figure 6.11 reveal that for most choices of δ, the SFL variants of Bit Vector Search, HyperSplit, and RFC always perform better than their non-

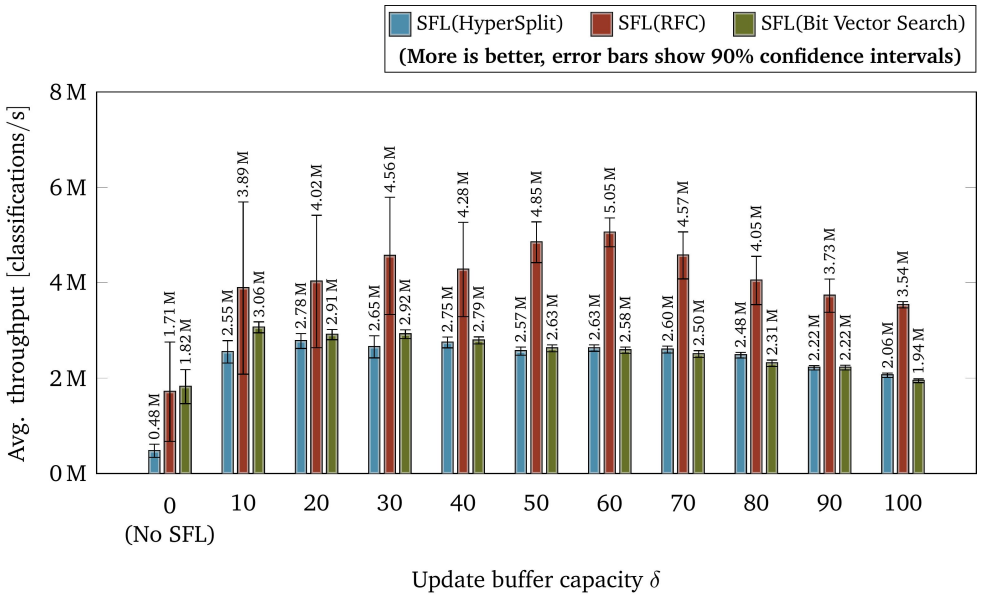


Fig. 6.10: Average classification throughput for varying update buffer capacities, with 100 issued rule insertions.

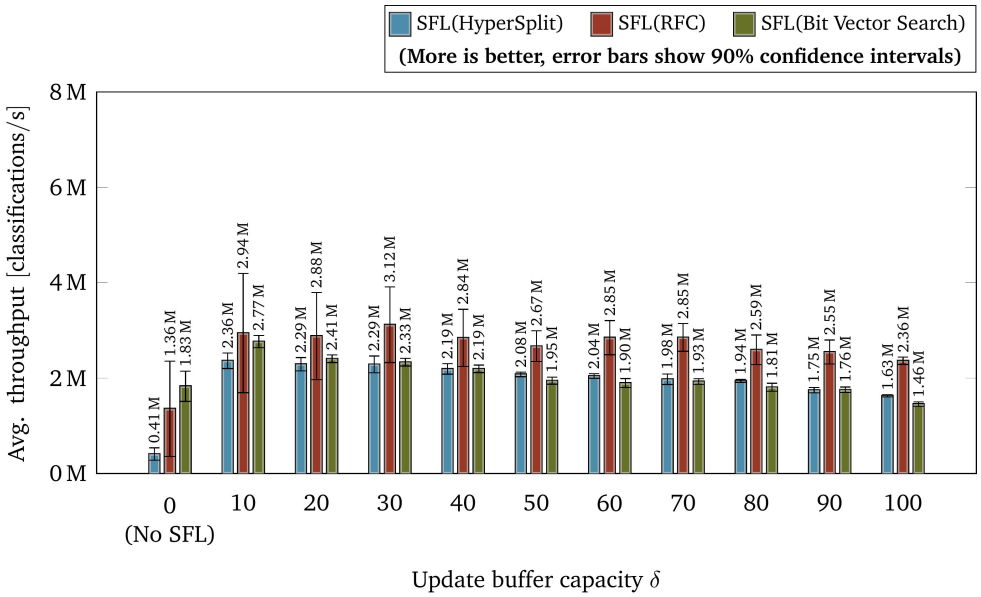


Fig. 6.11: Average classification throughput for varying update buffer capacities, with 100 issued rule deletions.

enhanced counterparts. However, the figures also show that choosing too large values for δ can result in decreasing throughput. This effect is especially well visible for the SFL(Bit Vector Search) algorithm, which reaches its peak performance at $\delta = 10$, performs worse for larger values of δ, and finally is outperformed by the non-enhanced Bit Vector Search for $\delta \geq 80$ in the deletion case. In contrast, the RFC algorithm performs best for larger values of δ, namely $\delta = 60$ in the insertion case and $\delta = 30$ in the deletion case. The results indicate that a good choice of δ depends on the utilized classification algorithm and also on the nature of the issued updates. For instance, in case of the RFC algorithm, larger values of δ compensate for the RFC search data structure creation, which is more expensive than those of the two other algorithms.

6.7 Limitations

The SFL classification system, as it is presented in this chapter, is designed to implement strict matching semantics with respect to the rule set installed in the system: any change to the rule set, when issued either manually by an administrator or through an automated controller, should take effect as soon as possible. This trait is useful in many situations, e. g., for rapid attack mitigation, frequent load balancing mechanisms, or environments with many short-lived rules. However, in systems that do not require such strict adherence to the master rule set and can tolerate certain delays until rule set changes take effect, the SFl approach is not required.

When considering a practical SFL implementation that goes beyond a prototype status, a better forcing mechanism than a static choice of the update buffer capacity δ is most likely required. Although we have shown the effects caused by different values for δ in our evaluation, we did not explain *how* the δ parameter should be chosen in practice. In fact, a practical implementation might consider to not even impose a capacity on the update buffer, but instead use a dedicated thread to constantly poll the update buffer for rule set changes. If the buffer is not empty, the thread could mark all nodes it the update buffer and compute the updated data structure in the background, while the update buffer nodes can be used together with the outdated search data structure until the new search data structure is ready to be used. When the data structure computation is complete, all marked update nodes could be removed from the buffer. This would simplify the system and also ensure that queued updates are quickly integrated into the fast search data structure used in the SFL classification process. Moreover, this approach would still guarantee strict matching semantics with respect to the master rule set, as updates are still initially installed in the update buffer.

Finally, it must be stressed that the SFL approach, as described in this chapter, is only able to solve the Complex Packet Classification Problem for stateless complex checks and terminal actions. This is due to the fact that the classification-relevant state could otherwise be changed by the execution of complex checks within the wrapped search data structure, which might not be correct with respect to the currently installed master rule set.

7 Summary

Many existing classification algorithms focus either on high updateability at the cost of classification performance, or provide high throughput but require expensive search data structures. In this chapter, we introduced two approaches to bridge the gap between these two extremes: namely (Aggregated) Jit Vector Search and the generic SFL classification system.

Our first contribution, the (Aggregated) Jit Vector Search approach, is a classification technique that builds upon the well-known (Aggregated) Bit Vector Search scheme [31, 76] and enhances it by dynamically generated one-dimensional machine code search trees, lookup tables for small dimensions, as well as SIMD instructions for operations on large vectors. Here, the main idea is to specialize one part of the generic Bit Vector Search data structure to machine code tailor-made for a specified rule set and the implementation of the other part to CPU-specific vector operations. We have demonstrated that, among a wide variety of existing classification algorithms, our proposed approach provides the best classification performance for small rule set sizes and performs close to the fastest existing classification algorithm for medium to large rule set sizes, at moderate memory requirements and preprocessing times. Despite its high performance, (Aggregated) Bit Vector Search, and with that, our enhancements, avoid the severe scalability issues of other best-in-class algorithms, namely RFC [62] and decision tree approaches [63, 122].

Our second contribution is the SFL classification system, which acts as a wrapper around an existing high-performance classification algorithm, such that it can be used in highly dynamic environments. By storing rule set updates in a buffer, the SFL system can postpone costly rebuilds of the wrapped algorithms and still provide high classification throughput that is correct with respect to the currently installed rule set. The key to these traits is the hybrid SFL classification algorithm, which utilizes the matching information from the fast but outdated data structure of the wrapped algorithm and corrects it by traversing the linear update buffer. In our evaluation we have demonstrated, that the SFL approach can significantly outperform existing algorithms with dynamic update capabilities as well as techniques that solely rely on fast but expensive search data structures, when used in dynamic environments. More specifically, we measured classification

throughput increases up to 4.8× and update throughput increase up to 4.5×, when we equipped the fastest existing classification algorithm with an SFL wrapper.

Part II

Rule Set Transformation

8 Introduction

In Part I, we saw a wide variety of sophisticated classification algorithms, that, when implemented at the core of a given classification engine, can significantly increase the system's packet throughput. However, many practically used and widely deployed classification systems [165, 167, 175] are still based on linear search and can easily be brought to their knees if the used rule sets grow in size [10, 12, 13]. In contrast, they provide powerful matching semantics and are extensible [166], which is of great importance for the definition of large and complex network topologies. For instance, at the time of writing, there exist over 80 publicly available extension modules for `iptables` [166], each of which comes with its own specific matching semantics. This makes it hard to natively integrate fast matching algorithms into the large code bases of these systems without introducing subtle bugs into existing and widely deployed functionality.

One suitable technique to improve the matching performance of existing classification systems without the need to change their implementation is to transform the rule set before it is used, as sketched in Figure 8.1. *Rule set transformation*, as introduced in Section 2.5, refers to the process of translating an input rule set $\mathfrak{R}$ to an output rule set $\mathfrak{R}'$, such that $\mathfrak{R}'$ heeds a desired transformation goal. Examples for such transformation goals are rule set size reductions, i. e., $|\mathfrak{R}'| < |\mathfrak{R}|$, or the removal of certain types of checks from the rule set in order to match the underlying system's matching capabilities [45, 114]. For most practical use cases [84], it is mandatory that the rule sets $\mathfrak{R}$ and $\mathfrak{R}'$ behave equally, i. e., are equivalent, with regard to their matching semantics: that is, the rule set transformation function t

The main algorithmic ideas for decision-tree-based rule set transformation presented in this chapter have been published as first author in [13] and [12]. The very first HiCuts-based prototype was subject of Stefan Selent's bachelor's thesis [24], as supervised by the author. The first prototype for a combination of decision-tree-based and reduction-based transformation approaches was created as part of Patrik John's diploma thesis [23] supervised by the author. Furthermore, the RuleBender approach in its entirety has been published as first author in [11].

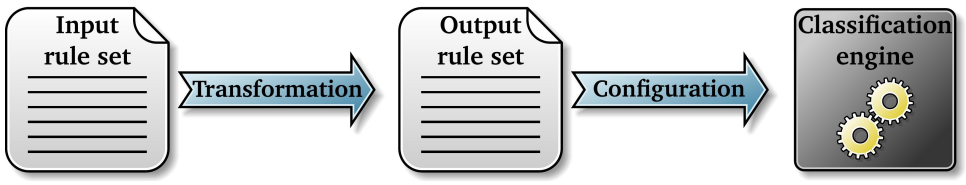


Fig. 8.1: Transforming a rule set before its installation in the system.

with $\mathfrak{R}' = t(\mathfrak{R})$ is required to be a semantics-preserving transformation function, as defined in Section 2.5.

Many rule set transformation schemes are technically orthogonal to the classification algorithm implemented in the underlying classification engine, in the sense that the transformation schemes do not require any knowledge about the internals of the utilized classification algorithm. This is especially true for generic *reduction-based approaches* that simply aim to reduce the number of rules in the specified rule set [49, 71, 84, 88], e. g., by removing redundant rules or merging different rules to a single rule. Therefore, generic reduction-based schemes are generally applicable to a large number of systems. However, these approaches come with two downsides: first, the slow classification performance of linear search-based systems is only mitigated, but not cured. Second, these techniques cannot deal with complex matching criteria, such as string matches in packet payloads, connection tracking, or custom user-defined match functions, which must be dealt with in order to solve the Complex Packet Classification Problem (CPCP). Instead, they focus solely on the compression of rules with simple numerical checks like port ranges or subnets, which impairs their usefulness for rule sets that utilize complex checks.

In order to bridge the gap between fast matching performance and existing classification systems with comprehensive matching capabilities, we propose the rule set transformation technique *RuleBender*. RuleBender brings the high lookup performance of decision tree classification algorithms [63, 107, 122] to existing packet processing systems *without the need to modify the systems' underlying implementation*. We exploit a widespread rule action feature that is implemented in many existing classification engines, namely the ability to execute fast *jumps* in the rule set. A jump is a special rule action that redirects the matching flow in the otherwise linearly traversed rule set if the rule defining the jump action matches the currently processed packet. Using jump rules, RuleBender encodes multi-dimensional decision trees in the generated output rule set. That is, although the underlying classification engine linearly probes the rule set for the first matching rule, RuleBender turns the slow linear search into a fast tree traversal. In contrast to existing minimization-based rule set optimization schemes [71, 84, 88], RuleBender expands the specified source rule set in a controlled way to integrate the decision tree search structures. This effort is rewarded by the fact that packets which are matched against RuleBender-generated rule sets traverse significantly fewer rules at run time when compared both to the original rule set and the compressed rule sets generated by [71], [84], and [88]. Furthermore, RuleBender supports match-based complex checks, which are transparently embedded in sub rule sets generated by the decision tree transformation. Therefore, RuleBender can be used with classification systems targeting the CPCP. Finally,

RuleBender can be selectively combined with reduction-based schemes to mitigate the output rule set expansion and to further increase the traversal performance of the generated rule set.

Our in-depth evaluation for rule sets up to 4,096 rules, which is based on the `iptables`, `nftables`, and `ipfw` packet filters, demonstrates that RuleBender-transformed rule sets can increase the achievable network packet throughput by up to 49× when compared to unmodified rule sets, and by up to 7× in comparison to reduction-based approaches [71, 84, 88]. Moreover, we examine RuleBender's adjustment screws in detail and describe a combined scheme with related work.

The remainder of this part is structured as follows: in Chapter 9, we discuss related work and depict the differences to RuleBender. Subsequently, Chapter 10 depicts the basic RuleBender approach as well as several enhancements. Also, the different RuleBender variants are evaluated in Chapter 10 and compared to the previously discussed related work. Finally, we conclude this part and summarize our findings in Chapter 11.

9 Related Work

Before we dive into the details of the proposed RuleBender algorithm, we first review existing rule set optimization techniques. We put our focus on existing static optimizers, as they are closest to the proposed RuleBender approach. In contrast to RuleBender, these approaches approaches aim to generated an output rule that is smaller than the input rule set set by removing redundant rules and/or fusing rules when possible. Accordingly, for a given input rule set $\mathfrak{R}$, the general rule set property these approaches aim for is

$$\pi_{\mathfrak{S}_{\text{MTU}}}(\mathfrak{R}, \mathfrak{R}') := |\mathfrak{R}'| < |\mathfrak{R}|, \tag{9.1}$$

with $\mathfrak{R}'$ being the transformed rule set. In some cases, however, Property (9.1) cannot be achieved and must be relaxed to

$$\pi_{\mathfrak{S}_{\text{MTU}}}(\mathfrak{R}, \mathfrak{R}') := |\mathfrak{R}'| \leq |\mathfrak{R}|. \tag{9.2}$$

For example, if the input rule set does not specify redundant rules, an approach based on the removal of redundant rules will not generate a smaller rule set while still being successful. Furthermore, the subset $K_{\text{MTU}} \subseteq \mathfrak{S}_{\text{MTU}}$ of transformable rule sets is mostly restricted to range-based, i. e., geometric rule sets for the approaches described in the remainder of this chapter.

9.1 Firewall Rule Optimization

The *Firewall Rule Optimization (FIRO)* [71] approach by Katić and Pale is a reduction-based transformation algorithm, which detects and removes two kinds of rules from a specified input rule set: *shadowed rules* and *directly downward redundant rules*. A rule R_i is said to be *shadowed* if there exists another rule R_j such that $j < i$ and if the geometric representation of R_i is completely covered by the geometric representation of R_j, i. e., $B(R_i) \subseteq B(R_j)$. A shadowed rule R_i can safely be removed from the rule set, because there exists no packet $p \in P_{\text{MTU}}$ for which R_i is the most highly prioritized matching rule.

The algorithmic techniques described in this chapter exclusively refer to existing related work by other authors. These techniques are depicted in the author's words to provide an overview and understanding of existing state-of-the-art, against which the author's work can be compared.

Runtime	Memory requirements	Output rule set size	Max. output rule set path length	Supports complex checks
$\mathcal{O}(d \cdot n^2)$	$\mathcal{O}(d \cdot n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	no

n: number of rules d: number of fields

Tab. 9.1: Firewall Rule Optimization performance characteristics.

Furthermore, a rule R_i is *directly downward redundant* if it has a successor rule R_{i+1} such that $B(R_i) \subseteq B(R_{i+1})$ and $a^i = a^{i+1}$, that is, both rules define the same action. In this case, R_i can safely be removed, as every packet p that matches R_i will also match R_{i+1}, which results in the same action.

The number of required rule comparisons by FIRO is quadratic in the number of rules n, since each rule R_i must be compared with every more highly prioritized rule $R_j, (j < i)$, in the worst case, and each rule comparison requires at most d subset checks. Furthermore, we mention that FIRO in its current form does not support complex checks, as it completely relies on subset inclusion semantics, which may not be possible for complex checks. Even worse, in the most general case, rule comparisons are undecidable [111], because a complex check can be an arbitrary program specified by the user. FIRO's performance characteristics and properties are summarized in Table 9.1.

9.2 Complete Redundancy Removal with FDDs

Although the FIRO approach depicted in the previous section is able to remove *some* redundant rules from a rule set $\mathfrak{R}$, it will not necessarily detect *all* redundant rules. For example, when looking at the geometric rule set representation in Figure 9.1, it is obvious that rule R_3 not entirely covered by neither R_1 nor R_2, but by their union, i. e., $B(R_3) \subseteq B(R_1) \cup B(R_2)$. As a consequence, there exists no packet $p \in P_{\text{MTU}}$ for which R_3 will be the most highly prioritized matching rule, because if R_3 matches p, then R_2 or R_1 will also match p. The *Complete Redudancy Removal (CRR)* approach [84] by Liu and Gouda is able to detect such so-called *upward redundant rules* by utilizing *Firewall Decision Diagrams (FDDs)*. Originally used for error detection in firewalls [59, 85], an FDD is a directed acyclic graph that represents all *effective rule sets* $\mathfrak{R}_i$ in a rule set $\mathfrak{R}$. For a given rule $R_i \in \mathfrak{R}$, the corresponding effective rule set $\mathfrak{R}_i$ describes all packets $p \in P_{\text{MTU}}$ matched by R_i, but not by any more highly prioritized rule R_j with $j < i$. Intuitively, the

geometric shape of $\mathfrak{R}_i$ is obtained by subtracting the geometric shapes of all more highly prioritized rules R_j from the geometric shape of R_i, that is,

$$B(\mathfrak{R}_i) = B(R_i) - \bigcup_{j=1}^{i-1} B(R_j). \tag{9.3}$$

Naturally, if $\mathfrak{R}_i = \emptyset$ for any rule $R_i \in \mathfrak{R}$, then R_i can safely be removed from $\mathfrak{R}$, because R_i can never be the most highly prioritized matching rule for any given packet.

In order to construct the effective rule sets for every rule R_i within a given rule set $\mathfrak{R}$, the CRR algorithm iteratively constructs an FDD by inserting one rule at a time into an initially empty decision diagram, beginning with rule R_1. The height of the constructed FDD is equal to the number of dimensions d, and every leaf node is labeled with a rule action. Furthermore, the FDD's edges are labeled with the set of possible packet headers that can follow a specific decision path in the diagram. For example, Figure 9.2a shows the FDD that includes only rule R_1. Because the path from node X to node a^1 was newly inserted, it represents R_1's entire effective rule set $\mathfrak{R}_1 = \langle R_1 \rangle$. When rule R_2 is inserted into the FDD, two new paths with leaf nodes labeled a^2 are created, as depicted in Figure 9.2b. Because the left highlighted path contains an edge labeled with a union of two disjoint intervals, the effective rule set $\mathfrak{R}_2$ contains three rules in total with

$$\mathfrak{R}_2 = \left\langle \begin{array}{l} ([4,4], [1,1], a^2), \\ ([4,4], [8,8], a^2), \\ ([5,7], [1,8], a^2) \end{array} \right\rangle, \tag{9.4}$$

which is also sketched by the dashed lines in Figure 9.1. The insertion of rule R_3 does not change the FDD from Figure 9.2b, because the recursive insertion process only traverses existing paths. Accordingly, $\mathfrak{R}_3$ is empty and is therefore marked as upward redundant.

Once the upward redundant rules have been removed from the input rule set $\mathfrak{R}$, $\mathfrak{R}$ may still contain so called *downward redundant rules*. A rule R_i is downward redundant, if for any packet $p \in P_{\text{MTU}}$, which is matched by R_i's effective rule set, there exists a less highly prioritized rule $R_{j,j>i}$ with $a^i = a^j$ that matches p, and no rule $R_k, i < k < j$ with $a^k \neq a^i$ that matches p. Hence, R_i can be removed because every packet, that would be matched by R_i, is subsequently matched by a less highly prioritized rule that exhibits the same action as R_i. The CRR approach detects downward redundant rules by again constructing an FDD, but this time it starts with the last rule. Then, for any inspected rule R_i, CRR analyzes whether the insertion of any rule in the effective rule set $\mathfrak{R}_i$ would create a new path in the FDD or overwrite the action in a leaf node. If this is not the case, then R_i

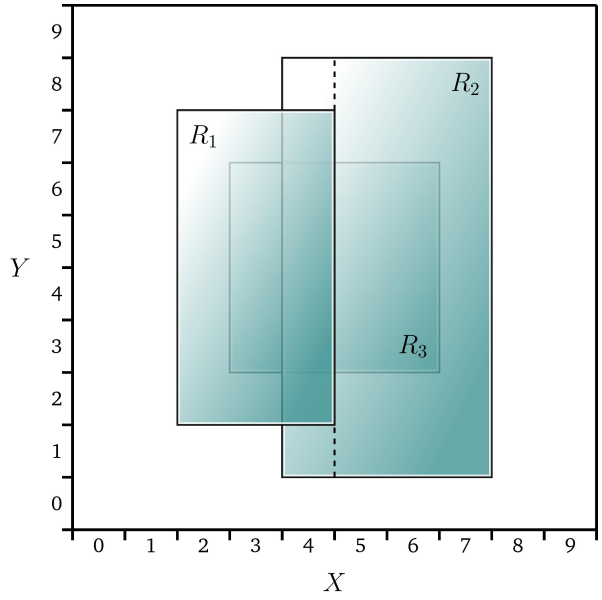
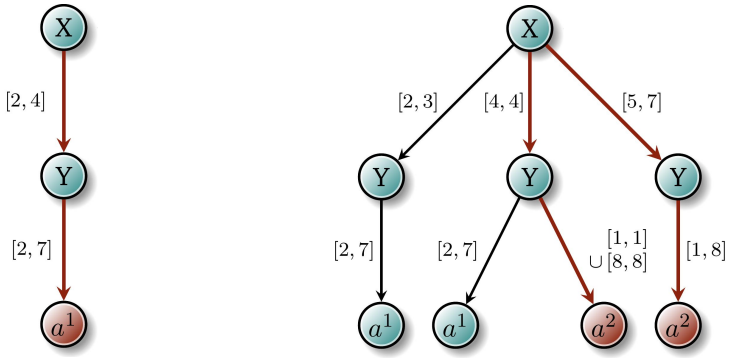


Fig. 9.1: Example for the redundant rule R_3, which is covered by R_1 and R_2. The dashed lines indicates one of two possible effective rule sets $\mathfrak{R}_2$ for rule R_2.



(a) FDD after inserting rule R_1.

(b) FDD after inserting rule R_2.

Fig. 9.2: Different stages of the incremental FDD construction process for the rules from Figure 9.1. The highlighted paths in the FDD represent the effective rule set $\mathfrak{R}_i$ after inserting rule R_i.

is downward redundant and can be removed, otherwise, R_i is inserted into the FDD.

In comparison to FIRO, CRR has higher storage and runtime requirements, as each inner node in the FDD can have $\mathcal{O}(n)$ children. Since the FDD has a height of $d+1$, its creation time and memory footprint are in $\mathcal{O}(n^d)$, as shown in Table 9.2. Similar to FIRO, CRR does not generally support complex checks, because the FDD data structure is entirely based on intervals, which requires checks to be represented as ranges.

Apart from redundancy removal, FDDs can serve for several different purposes: they are used in rule set verification and comparison [59, 85] and can, with

Runtime	Memory requirements	Output rule set size	Max. output rule set path length	Supports complex checks
$\mathcal{O}(n^d)$	$\mathcal{O}(n^d)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	no

n: number of rules d: number of fields

Tab. 9.2: Complete Redudancy Removal performance characteristics.

slight adjustments, be used as the primary search data structure in classification algorithms [115]. Moreover, they serve as the basic building block in several advanced rule set optimizers [49, 86, 87, 88], as we will see in Section 9.3 at the example of *Firewall Compressor* [88].

9.3 Firewall Compressor

The last rule set preprocessing algorithm we address in detail is *Firewall Compressor (FC)* by Liu, Torng, and Meiners [88]. Similar to the previously discussed CRR approach, FC also tries to reduce the number of rules in the given rule set. However, in contrast to CRR, FC aims to achieve this goal primarily through successive one-dimensional scheduling steps that are carried out on a so-called *Reduced Firewall Decision Diagram (RFDD)*, which essentially is an FDD without isomorphic sub trees. An example for the RFDD, which corresponds to the FDD from Figure 9.2b, is given in Figure 9.3.

The FC algorithm begins by transforming a given input rule set $\mathfrak{R}$ into an RFDD $T_{\mathfrak{R}}$. In the second step, the RFDD $T_{\mathfrak{R}}$ is traversed node by node in a bottom-up manner. Each inner node V in $T_{\mathfrak{R}}$ is regarded to represent a one-dimensional rule set, whose rules are assembled by the intervals of V's outgoing edges and the labels of the corresponding child nodes. More specifically, the intervals of V represent the one-dimensional rules' checks, while the labels of the child nodes are taken as the one-dimensional rules' actions. Once every inner node has been processed, the result rule set can be retrieved from the RFDD's root node. In the following, we sketch the inner workings of the complete FC algorithm by transforming the input rule set $\mathfrak{R}_{\text{input}} = \langle R_1, R_2 \rangle$ into an result rule set $\mathfrak{R}_{\text{res}}$.

For the example RFDD shown in Figure 9.3, the node V_3 represents the one-dimensional rule set

$$\mathfrak{R}_{V_3} = \left\langle \begin{array}{l} ([1,1]\,, a^2)\,, \\ ([2,7]\,, a^1)\,, \\ ([8,8]\,, a^2) \end{array} \right\rangle. \tag{9.5}$$

During the traversal of the RFDD, each node V's rule set $\mathfrak{R}_V$ is optimized using a one-dimensional scheduling algorithm that maps the rule set $\mathfrak{R}_V$ to an equivalent rule set $\mathfrak{R}'_V$, in the hope that $|\mathfrak{R}_V| < |\mathfrak{R}'_V|$. In case of the above example, the rule set $\mathfrak{R}_{V_3}$ is transformed into

$$\mathfrak{R}'_{V_3} = \left\langle \begin{array}{l} ([2,7]\,, a^1)\,, \\ ([1,8]\,, a^2) \end{array} \right\rangle, \tag{9.6}$$

which is equivalent to $\mathfrak{R}_{V_3}$, but contains only two instead of three rules. Analogously, the node V_1's rule set $\mathfrak{R}_{V_1}$ is

$$\mathfrak{R}_{V_1} = \left\langle \begin{array}{l} ([2,3]\,, V_2)\,, \\ ([4,4]\,, V_3)\,, \\ ([5,7]\,, V_4) \end{array} \right\rangle, \tag{9.7}$$

which, in this case, is equal to its optimized variant $\mathfrak{R}'_{V_1}$. As the node V_1 is the root of the RFDD, the rule set $\mathfrak{R}'_{V_1}$ represents the result of the FC algorithm, after each action V in $\mathfrak{R}'_{V_1}$ has been replaced by the rule sets in the corresponding child node V. Therefore, the resulting rule set $\mathfrak{R}_{\text{res}}$ is

$$\mathfrak{R}_{\text{res}} = \left\langle \begin{array}{l} ([2,3]\,, [2,7]\,, a^1)\,, \\ ([4,4]\,, [2,7]\,, a^1)\,, \\ ([4,4]\,, [1,8]\,, a^2)\,, \\ ([5,7]\,, [1,8]\,, a^2) \end{array} \right\rangle. \tag{9.8}$$

This example shows that the FC algorithm is not guaranteed to succeed in the goal of size reduction, because in this case, the output rule set $\mathfrak{R}_{\text{res}}$ has twice the amount of rules as the input rule set $\langle R_1, R_2 \rangle$. The authors of [88] propose that a subsequent redundancy removal using the CRR algorithm should be used in order to further reduce the size of the resulting rule set. However, in the case of $\mathfrak{R}_{\text{res}}$, no rule is upward or downward redundant, and hence no reduction takes place. In order to avoid a rule set size expansion, a possible strategy would be to return the original input rule set $\mathfrak{R}_{\text{input}}$, if $|\mathfrak{R}_{\text{res}}| > |\mathfrak{R}_{\text{input}}|$, as otherwise the targeted rule set property would not be achieved.

In terms of worst-case runtime and memory footprint, the FC approach exceeds the CRR algorithm, because an FDD needs to be constructed in the first step. Furthermore, each inner node requires the execution of the one-dimensional scheduling algorithm, which is linear in the number of rules. Accordingly, both Firewall Compressor's runtime and memory requirement is in $\mathcal{O}(n^{d+1})$, as summarized in Table 9.3. When it comes to matching criteria that cannot represented as ranges, FC suffers from the same problems as the FIRO and FIRO algorithms.

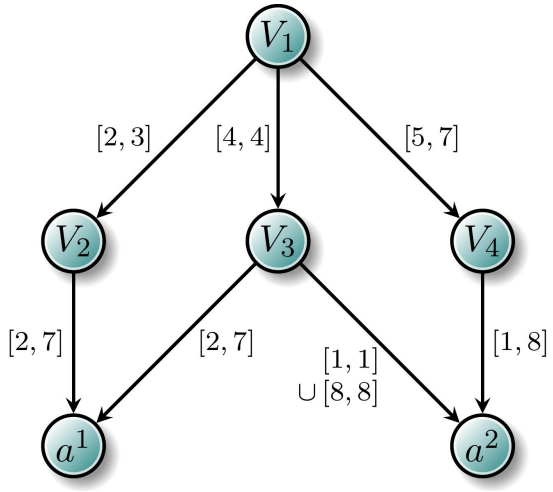


Fig. 9.3: RFDD for the FDD from Figure 9.2b.

Runtime	Memory requirements	Output rule set size	Max. output rule set path length	Supports complex checks
$\mathcal{O}(n^{d+1})$	$\mathcal{O}(n^{d+1})$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	no

n: number of rules d: number of fields

Tab. 9.3: Firewall Compressor performance characteristics.

9.4 Dynamic Rule Set Transformation

Up to this point, we reviewed *static* (or *offline*) rule set minimization techniques, i. e., techniques, that aim to programmatically reduce the number of rules in a specified rule set solely on information provided by the rule set itself. Typically, such static transformations are executed before the thereby optimized rule set is installed in the classification engine. In contrast, *dynamic* (or *online*) rule set transformation approaches utilize additional information only available at system run time, such as rule match counters [26, 65, 78, 130]. Here, the aim is to reorder rule in the installed rule set, such that the more frequently hit rules are placed near the top of the rule set and with the condition that the rule set semantics is maintained. Consequently, these approaches mainly target classification systems based on Linear Search, which allow rapid incremental rule set changes, while the system is active. Furthermore, much in the same manner as JITs [32], the operations performed by dynamic rule set transformation schemes are more time- and memory-constrained as those of static approaches, as dynamic transformations represent an additional computational burden on the classification system [65].

Although the proposed RuleBender technique takes a different approach to rule set optimization than the above dynamic schemes due to its static transformation, it

is generally possible to dynamically optimize the linear sub rule sets that represent the decision trees' child nodes.

10 The RuleBender Approach

In this chapter we introduce RuleBender, a rule set transformation technique to increase the search performance of linear-search-based classification systems. The motivation behind RuleBender is that the classification cost of a packet p mainly depends on the *classification path length*, i. e., the number of rules that are actually traversed by p—and not on the total number of rules in the rule set. This can be illustrated by a small example: consider a rule set $\mathfrak{R}$ of n rules, where the most highly prioritized rule R_1 is a wildcard rule that matches every possible packet $p \in P_{\text{MTU}}$. Although this is not a realistic use case, it is easy to see that the underlying classification system achieves optimal constant classification performance, because the search always terminates at the first rule for every incoming packet P, regardless of n.

Thus, RuleBender's goal is to transform a given input rule set $\mathfrak{R}$ into another rule set $\mathfrak{R}'$, such that the average classification path length is significantly reduced. Generally, if $T_{\mathfrak{R}}$ is a decision tree generated by HiCuts or HyperSplit, then the rule set property targeted by RuleBender is

$$\pi_{\mathfrak{S}_{\text{MTU}}}(\mathfrak{R}, \mathfrak{R}') := \mathfrak{R}' \text{ encodes } T_{\mathfrak{R}} \tag{10.1}$$

In order to achieve this goal, the RuleBender transformation consists of two phases: first, a decision tree search structure is generated for the input rule set $\mathfrak{R}$, which encodes a short classification path for incoming packets. Second, the decision tree is translated back into valid rule set syntax using jump rules that is understandable by the underlying classification system. As a result, incoming packets can be processed faster due to their significantly smaller classification path lengths exhibited by the decision trees encoded in $\mathfrak{R}'$. This entire process is sketched in Figure 10.1.

10.1 RuleBender Transformation

The first step of the RuleBender algorithm is to generate a decision tree search structure for the specified input rule set by using the preprocessing step of a suitable decision tree classification algorithm. In this context, the term "suitable" refers to decision tree algorithms that do not require to keep other matching

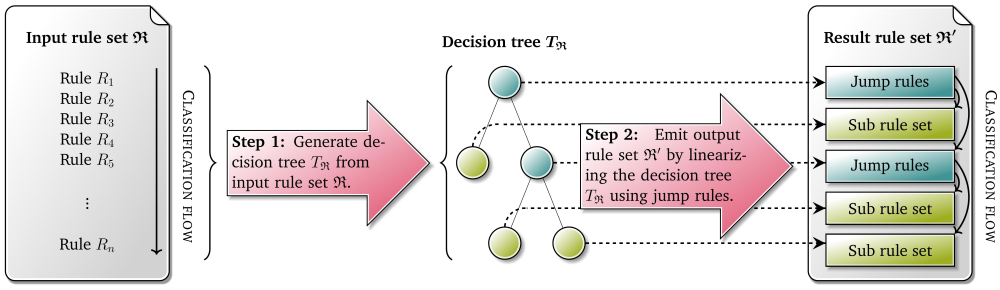


Fig. 10.1: Sketch of the RuleBender approach.

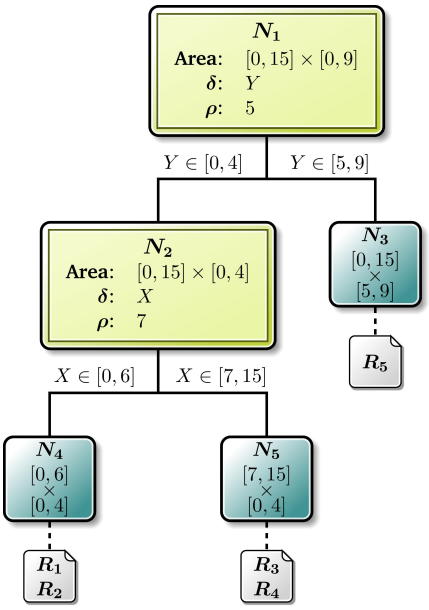
state than the currently traversed tree node. For example, HiCuts [63] or HyperSplit [107], as described in Section 4.5, are suitable candidates. In contrast, the EffiCuts [137] and HyperCuts [122] algorithms are not suitable because they create several decision trees with rules in non-strict order, which requires to remember the most highly prioritized matching rule for all the tree traversals. For the remainder of this section, we assume without loss of generality that the HyperSplit algorithm is the utilized suitable decision tree algorithm, if not specified otherwise. Also, we use the rule set $\mathfrak{R}$ shown in Figure 10.2a as a running example in order to illustrate the complete RuleBender transformation process. Because RuleBender utilizes the preprocessing step of the HyperSplit algorithm, the input rule set $\mathfrak{R}$ is used to generate the corresponding decision tree $T_{\mathfrak{R}}$ shown in Figure 10.2b.

The original decision tree algorithms, such as HiCuts [63] or HyperSplit [107], directly use the obtained decision tree search structures to classify incoming network packets. However, this assumes that the underlying classification engine natively implements the advanced classification algorithm, which is not the case for many widely used linear-search-based packet filters, such as FreeBSD's `ipfw` [165], Linux' `iptables` [167], or Linux' `nftables` [168]. Fortunately, these engines support and efficiently implement *jump rules* that can be exploited to linearize the decision tree and embed it in the generated output rule set, which is the second step of the RuleBender approach.

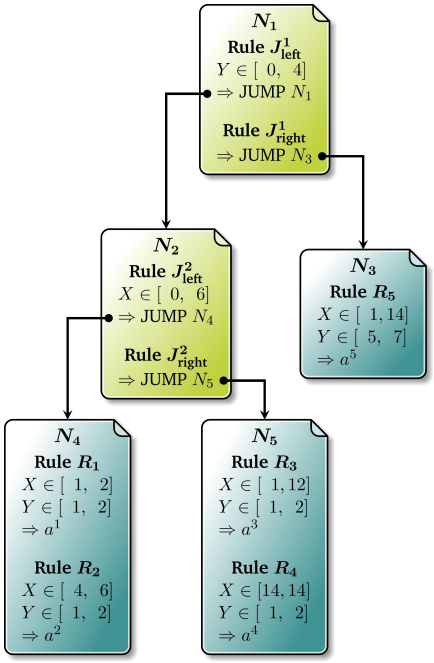
Therefore, in the second step, RuleBender traverses each tree node in pre-order in order to linearize the generated decision tree, starting at the root node N_1. If the currently regarded tree node N_i is an inner node, then RuleBender emits two *jump rules* J^i_{left} and J^i_{right}, one for the left branch and one for the right branch. The jump rule J^i_{left} encodes a range check $h^p_\delta \in [0, \rho - 1]$ on the dimension δ in which N_i's subspace has been cut at point ρ during the tree construction. In addition, J^i_{left} defines a jump action which redirects the matching flow to the subsequently defined sub rule set for N_i's left child node in case the range check matches h^p_δ. The jump rule J^i_{right} is defined similarly and redirects to N_i's right child, with

Rule ID	Dim. X	Dim. Y	Action
R_1	$X \in [1,2]$	$Y \in [1,2]$	a^1
R_2	$X \in [4,6]$	$Y \in [1,2]$	a^2
R_3	$X \in [8,12]$	$Y \in [1,2]$	a^3
R_4	$X \in [14,14]$	$Y \in [1,2]$	a^4
R_5	$X \in [1,14]$	$Y \in [5,7]$	a^5

(a) The input rule set $\mathfrak{R}$.



(b) The corresponding decision tree $T_{\mathfrak{R}}$.



(c) Decision tree sub rule sets.

Label	Rule ID	Dim. X	Dim. Y	Action
N_1:	R'_1 (J^1_{left})		$Y \in [0,4]$	JUMP N_2
	R'_2 (J^1_{right})			JUMP N_3
N_2:	R'_3 (J^2_{left})	$X \in [0,6]$		JUMP N_4
	R'_4 (J^2_{right})			JUMP N_5
N_4:	R'_5 (R_1)	$X \in [1,2]$	$Y \in [1,2]$	a^1
	R'_6 (R_2)	$X \in [4,6]$	$Y \in [1,2]$	a^2
N_5:	R'_7 (R_3)	$X \in [8,12]$	$Y \in [1,2]$	a^3
	R'_8 (R_4)	$X \in [14,14]$	$Y \in [1,2]$	a^4
N_3:	R'_9 (R_5)	$X \in [1,14]$	$Y \in [5,7]$	a^5

(d) The RuleBender result rule set $\mathfrak{R}'$.

Fig. 10.2: Example for a rule set transformation with RuleBender.

the difference that J^i_{right} is a wildcard rule that matches on every packet. This construction is correct, since packets that have not been sent to the left child must continue on the right child.

When the linearization process encounters a leaf node N_{leaf}, it emits the unchanged original rules from the source rule set, that have been assigned to N_{leaf},

in the correct relative order to preserve the source rule set's semantics. Furthermore, if the source rule set defines a default rule, a final wildcard *interception rule* is generated for N_{leaf} that specifies the rule set's default action. This is an important detail, because these rules intercept packets that do not match any rule in the rule set and prevent them from continuing further pointless traversal of the linearized tree, which can lead to wrong classification results.

The entire tree linearization is summarized by the TREE_TO_RULE_SET procedure in Algorithm 10.1, and the sub rule sets for the decision tree nodes as well as the resulting output rule set $\mathfrak{R}'$ for our running example are shown in Figure 10.2c and Figure 10.2d, respectively. The rules R'_1 to R'_4 in Figure 10.2d are jump rules that encode the dispatch logic for the inner nodes from the decision tree in Figure 10.2b. All remaining rules represent the decision tree's leaf nodes and thus are the original rules from the source rule set $\mathfrak{R}$. Note that $\mathfrak{R}$ does not define a default rule in our example, and therefore, the sub rule sets for the leaf rules do not specify interception rules.

Incoming packets start traversing the rules in $\mathfrak{R}'$ at the N_1 label. Note that, although the underlying engine uses Linear Search, we are still able to implement decision tree dispatch logic on top of it through the use of jump rules, which typically results in a significantly faster matching process. However, this comes at the cost of an increased rule set size: $\mathfrak{R}'$ contains 9 rules, while the original rule set $\mathfrak{R}$ from Figure 10.2a specifies only five rules. This expansion factor is greatly affected by the binth parameter β: a higher value of β leads to smaller decision trees, because the tree construction process terminates more quickly. Consequently, fewer jump rules must be emitted, at the cost of more expensive Linear Searches in the leaf nodes.

One important feature of RuleBender's decision tree preprocessing is that it can be applied to rule sets with complex match-based checks. That is, the set $K_{\text{MTU}} \subseteq \mathfrak{S}_{\text{MTU}}$ consists of complex rule sets, in which every complex check is restricted to the match-based property, as described in Section 2.4. Since we are able to build the decision tree solely on the basis of the rules' geometric shapes, we can simply ignore potential complex checks in the rules during the preprocessing phase. This does not change the output rule set's semantics, as the decision tree uses the simple checks only to quickly reduce the number of possibly matching rules, while for the final decision the full rules in the respective leaf are matched. Furthermore, the property that complex checks are match-based guarantees that the classification-relevant state changes only in those rules that match an incoming packet, which by definition of the decision tree semantics are the same as in a linear rule set traversal.

```
1  function LINEARIZE_TREE(Node N, Label λ)
2      EMIT_LABEL(λ)
3      if N is not a leaf node then
4          λ_left, λ_right ← GET_UNIQUE_LABELS()
5          δ ← GET_CUT_DIMENSION(N)
6          ρ ← GET_CUT_POINT(N)
7          EMIT_LEFT_JUMP_RULE(δ, ρ, λ_left)
8          EMIT_RIGHT_JUMP_RULE(δ, ρ, λ_right)
9          N_left, N_right ← GET_CHILDREN(N)
10         LINEARIZE_TREE(N_left, λ_left)
11         LINEARIZE_TREE(N_right, λ_right)
12     else
13         ℜ_N ← GET_RULE_SET(N)
14         for Rule R_i ∈ ℜ_N do
15             EMIT_RULE(R_i)

16 function TREE_TO_RULE_SET(Node N)
17     LINEARIZE_TREE(N, "N_1")
```

emit dispatch rules for the current node

continue traversal

rules in leaf nodes are emitted linearly

Algorithm 10.1: HyperSplit tree to rule set conversion.

This is a major advantage over existing reduction-based schemes, which rely on the elimination of redundant rules based on geometric shapes [71, 84, 88]. These approaches are not able to handle many types of complex checks, such as string matching, connection tracking, or custom user-defined matching functions, because these checks cannot be represented as plain intervals. Even worse, in the extreme case of arbitrary user-defined functions, e. g., using `iptable`'s queuing feature to shunt packets into the user space [166], the problem of rule redundancy is not decidable [111]. In contrast, the decision tree transformation used by RuleBender dispatches only on header fields that are used in virtually every packet filtering situation, such as IP addresses, port fields, and protocol types.

10.2 Practical Range Check Considerations

In the previous section, we argued that the key to decision tree linearization inside of the generated output rule sets is the usage of jump rules with range checks on certain header fields. However, when generating the output rule set for a specific packet classification engine, such as `iptables`, `nftables`, or `ipfw`, we have to deal with the issue that these engines do not directly support range checks on every field. Although source and destination port range checks between two ports `A` and `B` are natively supported by the syntax `A:B` or `A-B`, we also need to be able to express ranges between arbitrary IP addresses. While `iptables`

and `nftables` support checks for ranges between arbitrary IP addresses, e. g, `5.5.3.4` and `5.5.3.20`, by `-[src|dst]-range 5.5.3.4-5.5.3.20` in the case of `iptables`, `ipfw` does not directly provide this functionality. Instead, we have to break up the numeric IP address range into a list of consecutive subnets that cover the entire range. In the above case, the subnets are `5.5.3.4/30`, `5.5.3.8/29`, `5.5.3.16/30`, and `5.5.3.20/32`. These subnets are stored in a *table*, a fast Radix-tree-based search structure [165], which can then be used in jump rules as the desired range check by either `ip [from|to] "table(m)"`, where `m` is the number of the used table.

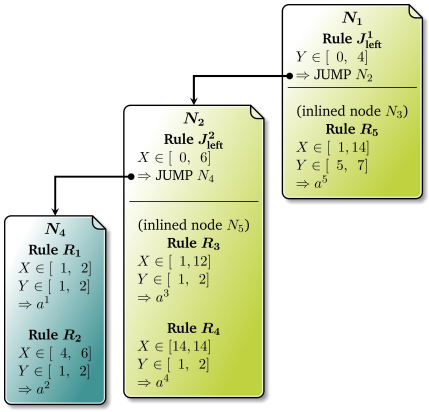
When it comes to sparsely populated fields without the possibility for range checks, such as protocol fields, we found that a reasonable strategy is to generate a separate decision tree for each specific value that is used in the source rule set. For example, a rule set $\mathfrak{R}$ that specifies rules for TCP, UDP, and ICMP packets, we first partition the rule set into three rule sets $\mathfrak{R}_{\text{TCP}}$, $\mathfrak{R}_{\text{UDP}}$, and $\mathfrak{R}_{\text{ICMP}}$. Subsequently, we generate a partial output rule set for each of these sub rule sets, which are concatenated. At the beginning of the concatenated rule set, we add a single jump rule for each sub rule set, such as

$$\begin{aligned} &\text{protocol} = \text{TCP} \quad \Rightarrow \quad \text{JUMP } \mathfrak{R}_{\text{TCP}} \\ &\text{protocol} = \text{UDP} \quad \Rightarrow \quad \text{JUMP } \mathfrak{R}_{\text{UDP}} \\ &\ldots \end{aligned} \tag{10.2}$$

Furthermore, we handle protocol wildcard rules, if they exist, by swapping them to the bottom of the rule set *before* the actual RuleBender transformation. Of course, in this case, each of the protocol-specific transformed rule $\mathfrak{R}_{\text{prot}}$ sets is constructed in a way that, if a packet cannot be classified within $\mathfrak{R}_{\text{prot}}$, it is redirected to the previously swapped-down wildcard rules. Finally, if a certain header field dimension δ can neither be queried by range checks nor is sparsely populated, we treat δ as a complex check dimension and do not regard it for the decision tree.

10.3 Enhancements of the Basic Scheme

RuleBender accelerates the classification performance of packet filters at the cost of increased sizes of the generated rule sets. This, in turn, has the undesired consequences of larger memory requirements and longer setup times when the rule set is installed in the packet filter engine. Therefore, we present two techniques to mitigate the rule set expansion factor induced by RuleBender: *branch inlining* and *a priori reduction*. These enhancements are not mutually exclusive and can be combined for greater effect.



(a) Inlined right branches.

Label	Rule ID	Dim. X	Dim. Y	Action
N_1:	R'_1 (J^1_{left})		$Y \in [0, 4]$	JUMP N_2
	R'_2 (R_5)	$X \in [1, 14]$	$Y \in [5, 7]$	a^5
N_2:	R'_3 (J^2_{left})	$X \in [0, 6]$		JUMP N_4
	R'_4 (R_3)	$X \in [8, 12]$	$Y \in [1, 2]$	a^3
	R'_5 (R_4)	$X \in [14, 14]$	$Y \in [1, 2]$	a^4
N_4:	R'_6 (R_1)	$X \in [1, 2]$	$Y \in [1, 2]$	a^1
	R'_7 (R_2)	$X \in [4, 6]$	$Y \in [1, 2]$	a^2

(b) The reduced result rule set.

Fig. 10.3: Inlining right branches during a RuleBender transformation.

10.3.1 Branch Inlining

Our first proposed extension of the basic RuleBender scheme, *branch inlining*, is motivated by the *function inlining* [44] compiler optimization technique. In the basic scheme, RuleBender emits two jump rules J^i_{left} and J^i_{right} for each inner node N_i during the output rule set generation phase. However, as explained in Section 10.1, the rule J^i_{right} is a wildcard rule that implements an unconditional jump to the sub rule set $\mathfrak{R}_{\text{right}}$ for the right child node. That is, the packet matching flow for a packet p *always* continues in $\mathfrak{R}_{\text{right}}$ if the left jump rule J^i_{left} does not match p.

Exploiting this structure, we can emit the rules in $\mathfrak{R}_{\text{right}}$ directly behind J^i_{left} instead of jumping to $\mathfrak{R}_{\text{right}}$. This modification is correct because the underlying classification engine linearly searches the rules and will thus continue the search in $\mathfrak{R}_{\text{right}}$, if it has not been redirected earlier by J^i_{left}. Branch inlining requires only minor modifications to the LINEARIZE_TREE procedure in Algorithm 10.1: the swapping of lines 10 and 11 as well as the omission of line 8. Figure 10.3 shows the HyperSplit sub rule set tree with inlined right branches and the resulting output rule set $|\mathfrak{R}'|$ generated by RuleBender for the input rule set from Figure 10.2a, when branch inlining is applied. As sketched in the figure, $|\mathfrak{R}'|$ is reduced from nine to seven.

Accordingly, the main result of branch inlining is the size reduction of the generated output rule set, because this technique halves the number of jump rules emitted during the tree generation process. However, another positive side effect is that fewer rules must be traversed and fewer jumps must be executed during the classification process, which results in a higher classification performance, as we demonstrate in Section 10.5.

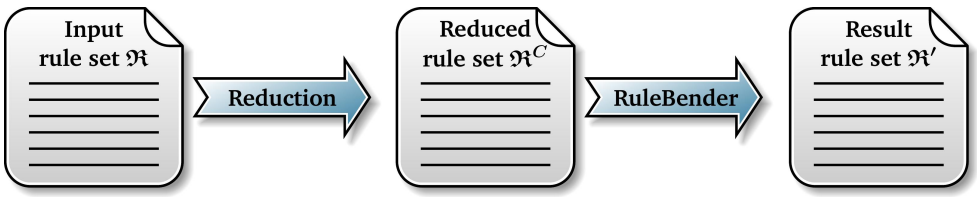


Fig. 10.4: Sketch of a priori reduction with RuleBender.

10.3.2 A Priori Reduction

RuleBender and the existing reduction-based rule set modification techniques *Firewall Rule Optimization (FIRO)* [71], *Complete Redudancy Removal (CRR)* [84], and *Firewall Compressor (FC)* [88] work towards the same goal, namely the generation of rule sets that can be traversed faster. However, RuleBender pursues a strategy that is opposed to the approach taken by reduction-based schemes: instead of reducing the number of rules, RuleBender enlarges the rule sets through decision tree encodings. This motivates an *a priori reduction* step through either FIRO, CRR, or FC *before* the RuleBender transformation. Accordingly, the computation of the output rule set $\mathfrak{R}'$ happens in a two-step process: in the first step, an intermediate compressed rule set $\mathfrak{R}^C$ is generated. After $\mathfrak{R}^C$ has been computed, we use it as the input rule set for RuleBender in order to generate the output rule set $\mathfrak{R}'$ in the second step. This procedure is sketched in Figure 10.4.

A priori reduction works without restrictions if the input rule set $\mathfrak{R}$ does not contain any complex rules. However, care must be taken if $\mathfrak{R}$ contains at least one complex rule, because neither FIRO, CRR, nor FC can deal with arbitrary complex rules. Therefore, in this case, we apply the following strategy: assuming that the rules at the indices $i_1, \ldots, i_k$ are complex, we extract the $k+1$ sub rule sets $\mathfrak{R}_1, \ldots, \mathfrak{R}_{k+1}$ that consist only of simple rules, i. e.,

$$\langle \underbrace{R_1, \ldots, R_{i_1-1}}_{\mathfrak{R}_1}, R_{i_1}, \underbrace{R_{i_1+1}, \ldots, R_{i_2-1}}_{\mathfrak{R}_2}, R_{i_2}, \ldots, R_n \rangle. \qquad (10.3)$$

Subsequently, we apply selective minimization to every sub rule set $\mathfrak{R}_j$ in order to compute the compressed sub rule sets $\mathfrak{R}_j^C$. The entire compressed rule set $\mathfrak{R}^C$, which is used as input for RuleBender, is then obtained by concatenating the sub rules sets $\mathfrak{R}_1^C \ldots \mathfrak{R}_{k+1}^C$ and the complex single-element rule sets $\langle R_{i_1} \rangle \ldots \langle R_{i_k} \rangle$ in the correct order by

$$\mathfrak{R}^C = \mathfrak{R}_1^C \langle R_{i_1} \rangle \ldots \langle R_{i_k} \rangle \mathfrak{R}_{k+1}^C. \qquad (10.4)$$

As demonstrated in Section 10.5, the benefits of selective minimization, when applied prior to RuleBender, are twofold: first, the size of the output rule set $\mathfrak{R}'$ is significantly reduced, in some cases by an order of magnitude. Second, this size reduction often leads to more efficient packet classification.

Algorithm	Runtime	Memory requirements	Output rule set size	Max. output rule set path length	Supports complex checks
Related work					
FIRO [71]	$\mathcal{O}(d \cdot n^2)$	$\mathcal{O}(d \cdot n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	no
CRR [84]	$\mathcal{O}(n^d)$	$\mathcal{O}(n^d)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	no
FC [88]	$\mathcal{O}(n^{d+1})$	$\mathcal{O}(n^{d+1})$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	no
Proposed approach					
RuleBender	$\mathcal{O}\left(\sum_{i=1}^{k} \lvert T_i \rvert\right)$	$\mathcal{O}\left(\sum_{i=1}^{k} \lvert T_i \rvert\right)$	$\mathcal{O}\left(\sum_{i=1}^{k} \lvert T_i \rvert\right)$	$\mathcal{O}\left(\max T_i^{\uparrow}\right)$	yes (match-based)

$T_i, i \in \{1, \ldots, k\}$: the HC/HS decision trees $T_i^{\uparrow}$: height of T_i (assumed in $\mathcal{O}(d)$ (HC)/$\mathcal{O}(d \cdot \log(n))$ (HS))
$\lvert T_i \rvert$: number of nodes in T_i (assumed in $\mathcal{O}(n^d)$) n: number of rules d: number of fields

Tab. 10.1: RuleBender performance characteristics, in comparison with related work.

10.4 Performance Characteristics

After having discussed the basic RuleBender scheme as well as its enhancements, we take a look at RuleBender's performance characteristics. The time it takes to transform an input rule set $\mathfrak{R}$ into an output rule set $\mathfrak{R}'$ is dictated by the time of the decision tree creations. Therefore, assuming that k decision trees are created (e. g., for different protocols, as described in Section 10.2), the transformation time and the memory footprint are in $\mathcal{O}\left(\sum_{i=1}^{k} |T_i|\right)$, where the T_i are the generated decision trees and $|T_i|$ denotes the number of nodes in T_i. The same holds for the size of the generated output rule set $\mathfrak{R}'$, which directly depends on the size of the generated decision trees. Of course, the transformation time increases correspondingly, when a priori reduction is applied, but since the asymptotic runtime of FIRO, CRR, and FC, are in $\mathcal{O}(n^d)$, the asymptotic runtime of RuleBender does not increase in these cases. The same reasoning can also be applied to the other performance characteristics under consideration.

When it comes to the maximum classification path length for an incoming packet, we can say that it is bounded by the height of the largest decision tree, plus a constant number of rules for the tree dispatch and the rules in the leaf nodes. As previously mentioned in Section 4.5, we assume that the height of HiCuts/HyperSplit trees to be in $\mathcal{O}(d)/\mathcal{O}(d \cdot \log(n))$, and the number of nodes in the trees to be in $\mathcal{O}(n^d)$, as suggested by the literature [58, 61, 107, 144, 147]. These performance characteristics are summarized in Table 10.1, in comparison to all existing static transformation approaches that were depicted in Chapter 9.

10.5 Evaluation

In this chapter, we evaluate RuleBender by comparing it with the existing FIRO [71], CRR [84], and FC [88] approaches on the basis of the four most important key performance indicators for rule set transformation schemes: the *transformation time*, the *rule set size expansion factor*, the mean *classification path length*, and the resulting achievable *classification throughput* when using the transformed rule sets. Furthermore, we examine each of the proposed RuleBender modifications in detail, and additionally investigate the influence of the decision tree binth parameters as well as the number of different actions on the quality of the generated output rule sets.

10.5.1 Experiment Setup

In order to carry out our experiments, we use our C implementations of RuleBender, FIRO, CRR, and FC. Also, we employ the widely used [43, 13, 69, 92, 107] *ClassBench* packet classification benchmark [132] to generate rule sets of sizes between 64 and 4,096 rules, in steps of 2^i with $i \in \{6, 8, 10, 12\}$. For each size, we generate ten different rule sets using ClassBench's *Access Control List (acl1)* rule set template. Each ClassBench-generated rule set consists of rules that define subnet checks on source and destination IPv4 addresses, the transport protocol (which is either TCP, UDP, or unspecified), and port fields. ClassBench is also used to generate a uniformly distributed header trace of 20,000 headers for every rule set.

Our evaluation setup consists of three computers: a *sender* machine with an Intel Xeon E5-1660 3.3 GHz CPU with eight physical cores (Hyper-Threading disabled) and 128 GB of RAM running Ubuntu Linux 17.04 Server, as well as two *firewall* machines equipped with a quad-core Intel Celeron 1.6 GHz CPU and 8 GB of RAM. One firewall machine runs Ubuntu Linux 17.04 Server with `iptables 1.6.0` and `nftables 0.6`, the other firewall machine runs FreeBSD 11.0 with `ipfw`. The sender is directly connected to each firewall machine via two 1 Gbit/s Ethernet links, where the first link is used to send traffic from the sender to the firewall machine, and the second link is used to relay all processed packets back to the sender. Accordingly, the firewall machines' routing tables are configured to directly forward each incoming packet back to the sender machine on a different interface. That way, we can evaluate the classification throughput of the firewall receiver machines by counting the number of packets received back on the corresponding sender interface. This evaluation setup is illustrated in Figure 10.5.

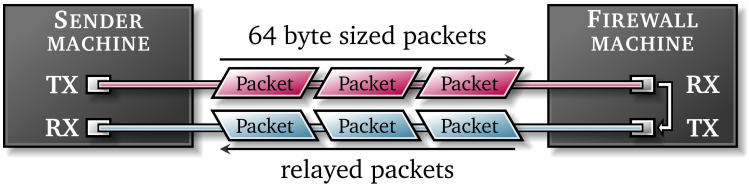


Fig. 10.5: Sketch of the RuleBender evaluation setup, showing the sender and one firewall machine.

Our evaluation code was compiled on the sender machine using `gcc 6.3.0` with the compile options `-Wall -Wextra -pedantic-errors -Werror -std=gnu99 -march=native -O3 -DNDEBUG`. The rule set transformations are executed on the sender machine, which also generates and sends the traffic of 64 byte sized TCP and UDP packets corresponding to the traces generated by ClassBench. We use the `tcpreplay` [176] tool to send the packets as fast as possible over a time period of ten seconds by looping over the trace to either the Linux or FreeBSD firewall machine. The firewall machines' packet filters are configured with the rule set under test, and after each test run, we extract the number of packets that were processed during ten seconds using the `netstat` tool. The last rule in every rule set is a match-all rule with an ACCEPT action. We evaluate the rule sets with two, four, and eight different actions ACCEPT_i, where each action is a redirection to ACCEPT (implemented via jumps). As ClassBench does not generate these actions, we take the generated rule sets and randomly distribute the actions over the rules within the rule set. These redirections are used in order to study the effect of different numbers of actions, without having to use real actions like DROP or REJECT, which would distort our measurement results. Hence, every rule in the source rule sets defines a randomly chosen action in $\{\text{ACCEPT}_i | i < i_{\text{max}}\}$, with $i_{\text{max}} \in \{2, 4, 8\}$. This is important for meaningful benchmarks, since the CRR and FC approaches would reduce every rule set, where each rule defines the same action, to a one-rule output rule set, which is not a realistic use case.

In the remainder of this section, the data points in the plots show the mean result of ten evaluation runs (each run with a different randomly generated rule set) and with corresponding 95% confidence intervals, if not stated otherwise in the plot captions. The transformation algorithms under scrutiny in are shown in Table 10.2. If not stated otherwise, the binth parameter β for the decision trees, as described in Section 4.5, is set to 16, because smaller values of β often lead to significantly longer preprocessing times. We deliberately choose to not cover the entire crossproduct of possible algorithm combinations, but instead focus on a meaningful subset of techniques that implement different algorithmic idea and yield a measurable improvement. For example, we do not cover HiCuts with inlined right branches, as it yields worse results than HyperSplit with inlined right branches and utilizes the same algorithmic enhancement.

Transformation algorithm	Abbreviation
Firewall Rule Optimization	FIRO
Firewall Rule Optimization	CRR
Firewall Compressor	FC
RuleBender with HiCuts	HC
RuleBender with HyperSplit	HS
RuleBender with HyperSplit and inlined right branches	HS (inline)
RuleBender with HyperSplit, inlined right branches, and a priori CRR	CRR $\rightarrow$ HS (inline)
RuleBender with HyperSplit, inlined right branches, and a priori FC	FC $\rightarrow$ HS (inline)

Tab. 10.2: Evaluated transformation algorithms.

In order to verify the correctness of the transformed rule sets, we use a self-written linear-search-based interpreter that matches a trace of packet headers against a specified rule set and logs the sequence of actions determined for the different packet headers. We consider a transformed rule set $\mathfrak{R}'$ to be correct, if it yields the same sequence of actions as the original input rule set $\mathfrak{R}$. This sanity check is done for every input rule set/output rule set combination and for every evaluated algorithm, and passes in every case. Furthermore, we use the interpreter to determine the average classification path length, i. e., the number of rules a packet header traverses until a final classification verdict can be issued.

10.5.2 Rule Set Size and Transformation Time

In the beginning of our evaluation, we take a look at the size of the rule sets generated by the different transformation techniques. Furthermore, we investigate the time it takes to transform an input rule set into a corresponding output rule set. In Figure 10.6, the sizes of the output rule sets for the different algorithms are shown. As expected, the application of the reduction based approaches FIRO, CRR, and FC leads to a smaller average output rule set size, while the RuleBender variants HC, HS, and HS (inline) inflate the rule set size, up to a considerable factor of 20.3$\times$ in the case of HiCuts with 4,096 rules. The large error bars in case of HyperSplit originate from one particularly large output rule set with about ten times as many rules as the other output rule sets, which can be explained by a series of unfavorable cuts during the tree construction process. However, we also observe that the rule set expansion is reduced when applying HyperSplit instead of HiCuts and through the usage of branch inlining. This can be explained by the fact that HyperSplit generally produces smaller decision trees

than HiCuts. Also, the HyperSplit trees are binary trees, that can be directly translated into two dispatch rules (or one, in the case of inlining). In contrast, the k-ary nodes produced by HiCuts must be transformed into binary search trees, which in turn requires more output rules. Lastly, we observe that a priori reduction can significantly further reduce the size of the result rule sets, especially for the larger rule set sizes, because HyperSplit generates smaller decision trees due to the reduced rule set size.

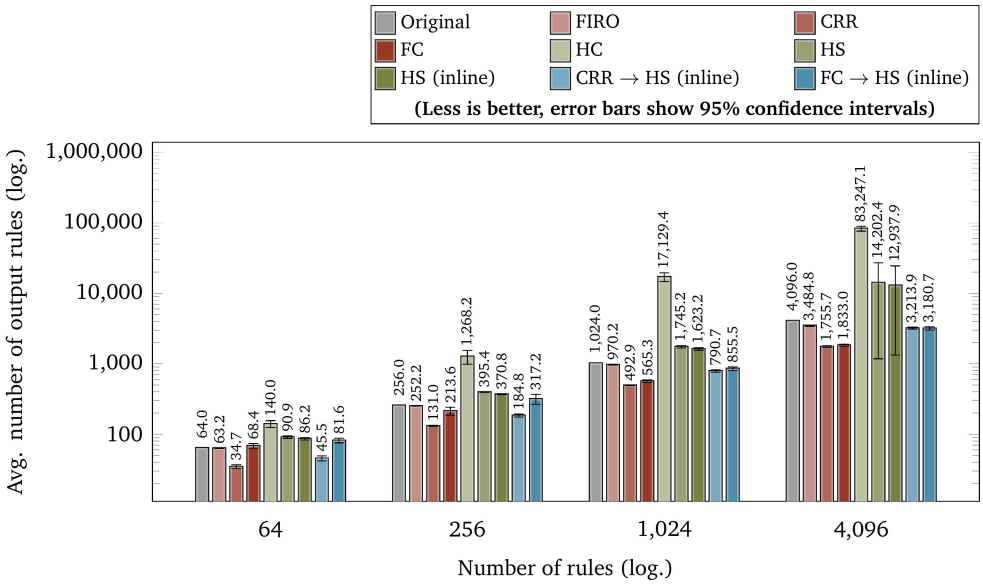


Fig. 10.6: Average output rule set sizes for the different transformation algorithms for rule sets with two actions. The exact size of the input rule sets are shown for reference purposes.

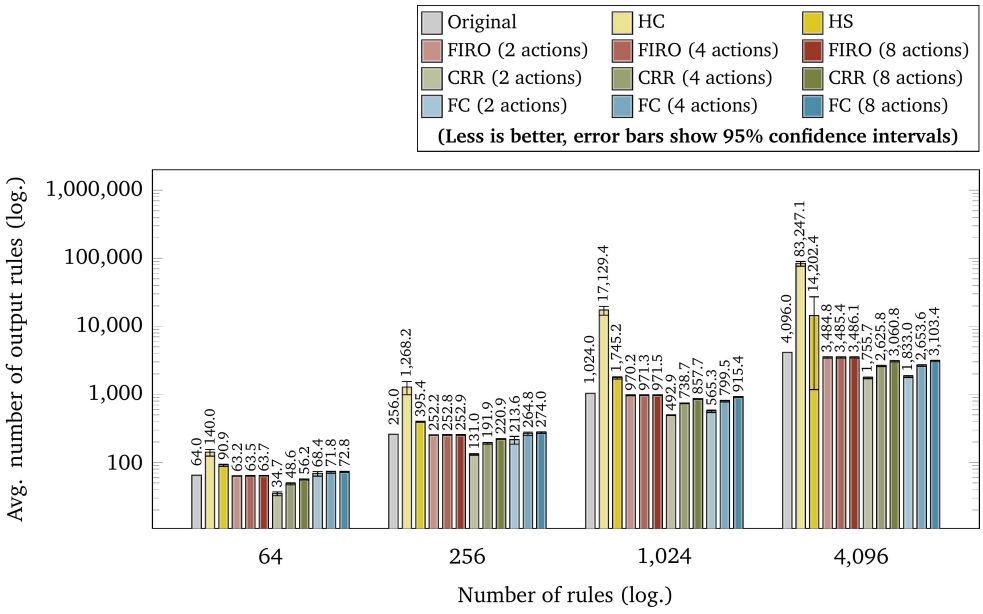


Fig. 10.7: Average output rule set size for the different transformation algorithms, depending on the number of actions in the input rule set.

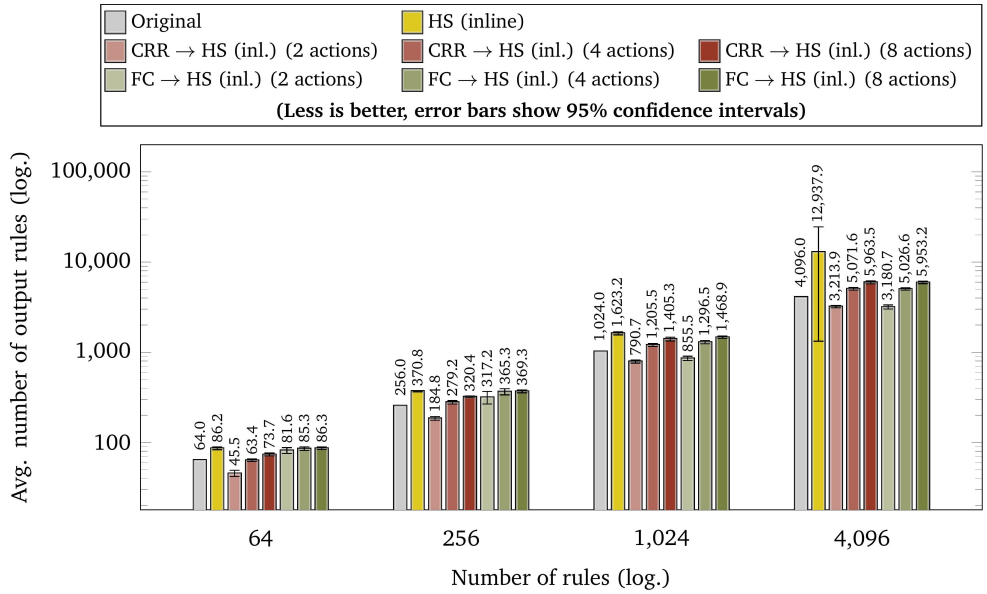


Fig. 10.8: Average output rule set size for RuleBender with HyperSplit (inline), depending on the number of actions in the input rule set.

Figure 10.7 demonstrates the effect of an increased number of actions on the effectiveness of reduction-based approaches. While HiCuts and HyperSplit are not affected by the number of different actions in a rule set, since the decision tree construction is action-agnostic, this is not the case for the removal of downward redundant rules, as performed by FIRO, CRR, and FC. Accordingly, the quality of RuleBender generated rule sets does not diminish with an increasing number of actions. However, the effectiveness of a priori reduction is of course also negatively impacted by an increasing number of actions, as confirmed by Figure 10.8. In fact, the more actions are used in the input rule set, the more the size of the rule set generated by RuleBender (inline) with a priori reduction converges against the result of plain RuleBender (inline).

Next, we investigate the transformation times for the different algorithms, which are shown in Figure 10.9. It can be seen that all algorithms can produce an output rule set in well below one second for the given rule sets, with FIRO being the fastest approach, since its runtime is only quadratic in the number of rules. Furthermore, we observe that plain RuleBender executes significantly faster than CRR and FC, especially when used with HyperSplit, as it yields smaller decision trees than HiCuts. The longest transformation time is required by RuleBender with a priori reduction, which is no surprise, since it has to execute two preprocessing algorithms instead of one.

Summarizing the investigation of output rule set sizes and transformation times, we note that all regarded algorithms perform reasonably fast for the utilized rule sets, and are suitable for environments that do not require several rule set

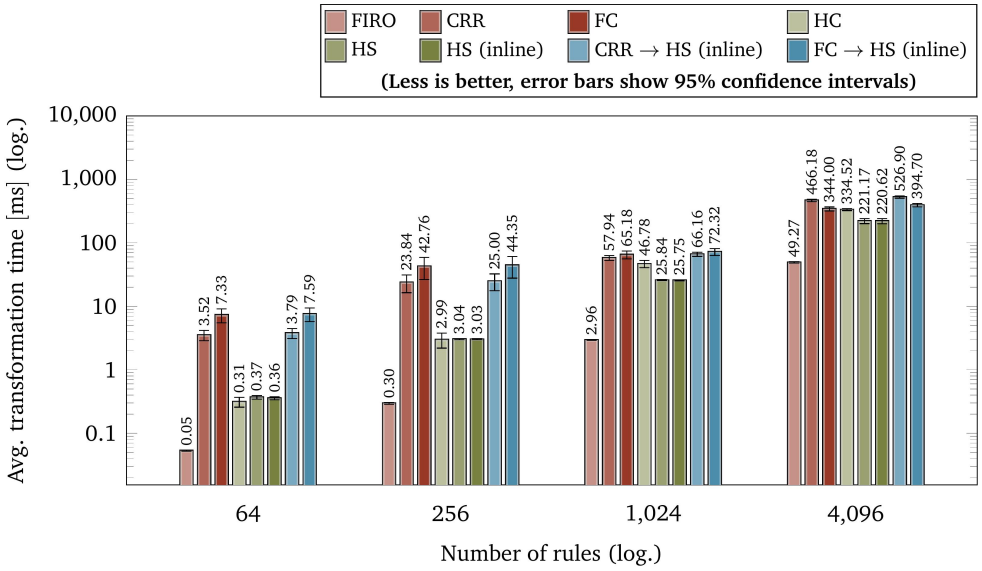


Fig. 10.9: Average transformation times for the different transformation algorithms for rule sets with two actions.

changes per second. Moreover, the results demonstrate the inflation character of RuleBender: in order to encode the decision trees in the rule set, the output rule set typically grows in size. This effect, however, can be diminished by the application of a priori reduction, whose effect increases with a decreasing number of actions. In fact, when the number of actions is only as small as two, the RuleBender-generated rule sets are in many cases even smaller than the input rule sets.

10.5.3 Path Length, Throughput, and Tree Height

As previously mentioned in Chapter 10, the main motivation behind RuleBender is the reduction of the classification path length for incoming network packets through the encoding of decision trees in the generated rule set. The shorter the average classification path length is, the less rules must be processed in the linear search executed by the underlying classification engine. Consequently, we expect that this results in a higher achievable firewall throughput.

Figure 10.10 shows the average classification path lengths for the different transformation approaches. The plot indicates that the reduction-based FIRO, CRR, and FC algorithms are able to reduce the mean path, in case of FC with 4,096 rules down to 40% in comparison to the unmodified rule sets. Furthermore, it can be observed that the RuleBender variants without a priori reduction significantly reduce the path lengths further for each of the rule set sizes under scrutiny, as a result of the rule-set-encoded decision trees. Here, RuleBender with HyperSplit

and inlined right branches yields the best results for every rule set size. This is particularly well visible in the case of 4,096 rules, where the average path length is reduced to 0.8% compared to the original rule sets' path lengths. Also, we see that RuleBender with a priori reduction performs insignificantly worse than RuleBender without this technique. The reason for this effect is the removal of downward redundant rules by CRR and FC: as explained in detail in Section 9.2, a rule R_i is downward redundant if a combination of less highly prioritized rules with the same actions cover R_i's effective rule set, without being disturbed by intermediate intersecting rules with different actions. However, the removal of downward redundant rules can occasionally lead to longer path lengths, which happens in the linear leaf rule sets in the encoded decision trees.

In the previous section, we saw that the number of different actions in the source rule sets influences the size of the output rule sets of reduction-based approaches. Figure 10.11 shows that this circumstance leads to longer path lengths for the FIRO, CRR, and FC algorithms, due to the greater size of the transformed rule sets.

Finally, we investigate the achievable throughput with actual network traffic when deploying the different rule sets onto our firewall machines, which are equipped with the `iptables` and `nftables` or `ipfw` packet filters, respectively. Figures 10.12, 10.13, and 10.14 show the results of our throughput experiments for the different rule set sizes and the different algorithms. On each system, we additionally show the achievable throughput for firewalls with an empty rule

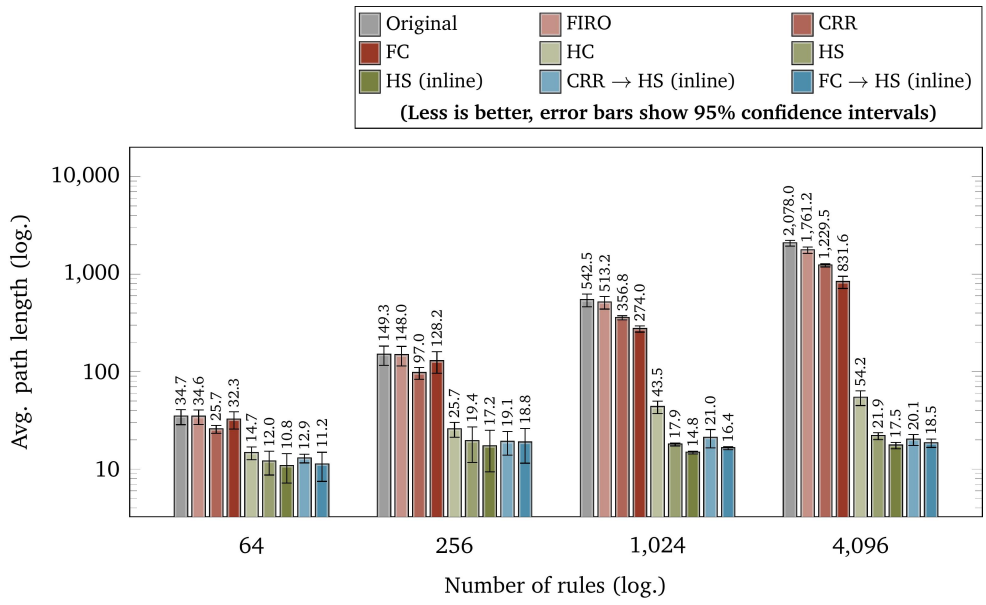


Fig. 10.10: Average classification path lengths for the different transformation algorithms for rule sets with two actions.

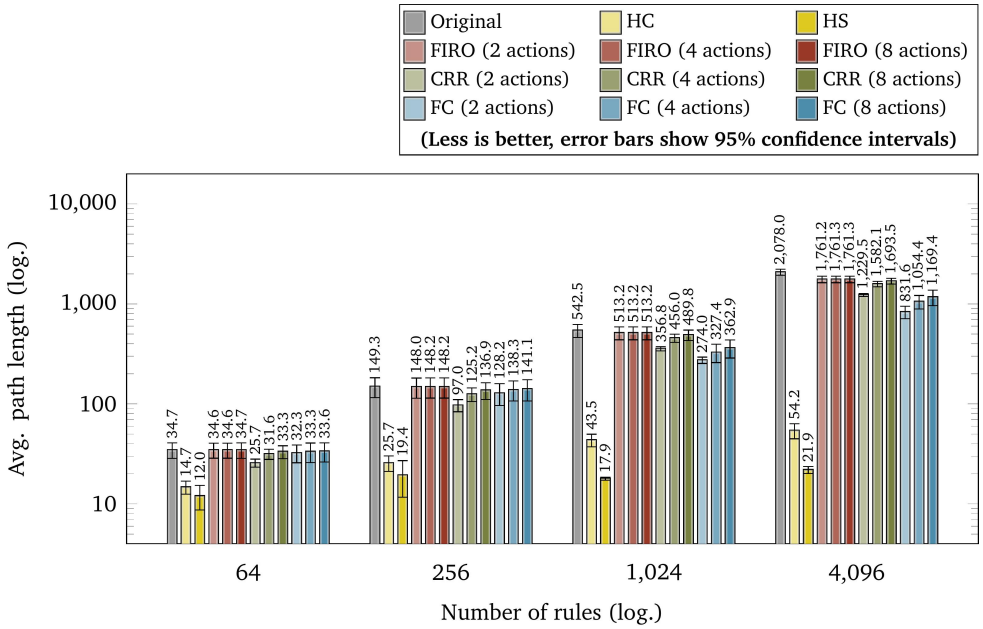


Fig. 10.11: Average classification path lengths for the different transformation algorithms, depending on the number of actions in the input rule set.

set for reference purposes, which represents an upper limit for the maximum classification performance on the respective system. First, we see that the Linux machine can process about 420 K packets per second, while the FreeBSD system only processes about 174 K packets per second. Unfortunately, we cannot explain where this discrepancy originates from, as it has many possible causes, such as kernel configurations, device drivers, or implementation details in the network stack. In both systems, the routing table is configured with as little as four entries and therefore unlikely to result in a severe bottleneck.

Moving on to the actual classification throughputs, it can be seen that for all three packet filters, the measured throughput drops drastically with an increasing number of rules in the firewall, down to about 3,500 packets per second in the case of `nftables` with 4,096 rules. Interestingly, `iptables` demonstrates a by far better performance than `nftables`, despite the fact that `nftables` is the de facto successor system to `iptables`. In all three cases, the application of reduction-based transformations typically leads to throughput increases, in the case of `nftables` with 4,096 rules up to a factor of 7×. However, the figures also demonstrate that the different RuleBender variants significantly outperform both the original rule set as well as the generated rule sets by FIRO, CRR, and FC, for the rule sets with a size greater than 64 rules. The only exception here is RuleBender with the HiCuts decision tree algorithms, whose search trees are much larger than those utilized by HyperSplit. This is confirmed by Figure 10.15, which illustrates the average maximum tree heights for both HiCuts and HyperSplit with different binth parameters. It can be seen that for a binth value of 16,

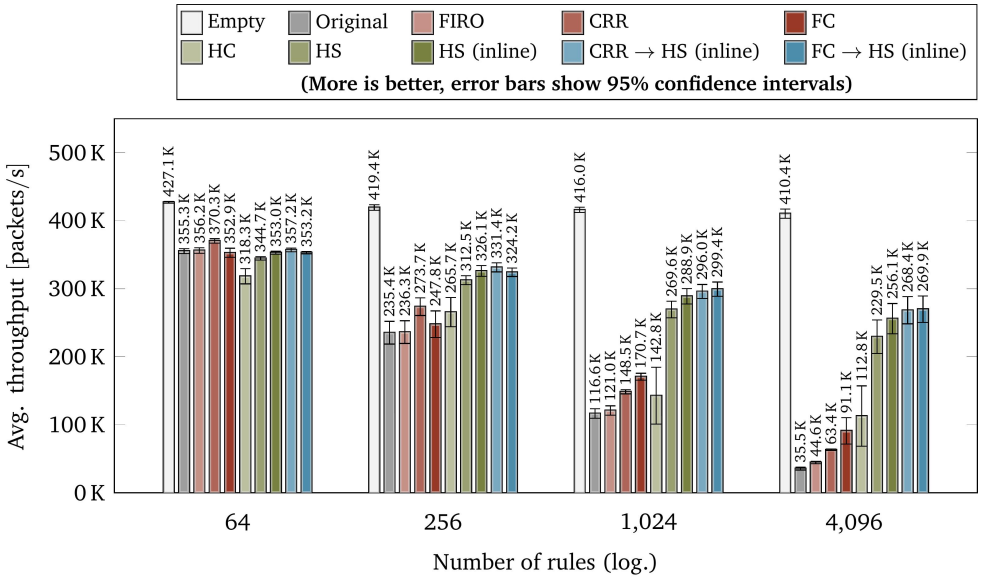


Fig. 10.12: Average `iptables` firewall throughput for rule sets generated by the different transformation algorithms, for input rule sets with two actions. The throughput for an empty firewall is shown for reference purposes.

which we used for our experiments, the HiCuts tree depths are significantly larger than for HyperSplit. Furthermore, we must keep in mind that the application of HiCuts requires the generation of a binary search tree per cut step, which further deepens the total tree height. This is also the reason why we could not evaluate RuleBender with HiCuts on the `nftables` firewall, because `nftables` rule sets have a hardcoded limit for the number of consecutively linked rule chains, which is 15. Since we do a transport layer protocol dispatch in the beginning of the transformed rule sets, as described in Section 10.2, and furthermore have to re-map our virtual ACCEPT$_i$ actions to actual ACCEPTs, we are limited to a maximum tree height of 13, as shown in Figure 10.15. This limitation also forced us to increase the binth for HyperSplit from 16 to 32 or 64 in some cases.

Nevertheless, we still achieve the best throughput increase for RuleBender on the `nftables` system, which is up to factor of 44.2× without a priori reduction, when compared to the original rule set, and up to a factor of 6.3×, when compared to the best results of reduction-based approaches. With a priori reduction, we can further increase these factors to 49.7× and 7.1× with `nftables`, respectively. We achieve qualitatively similar results with `iptables` and `ipfw`, but the factors look different here: with `iptables`, RuleBender without a priori reduction increases the throughput up to factor of 7.2×, and with a priori reduction to 7.6× when compared to the original rule sets. When compared to the best related work results with the FC algorithm, as indicated by Figure 10.12, these factors are 2.8× and 2.9×, respectively. In the case of `ipfw`, these factors are 13.7× (RuleBender without a priori reduction vs. original), 14.1× (RuleBender with a

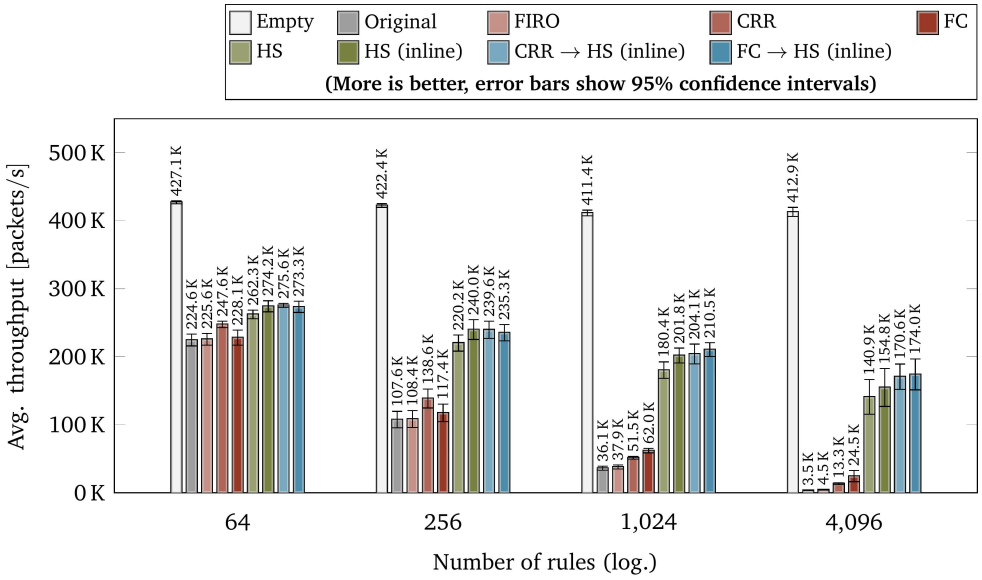


Fig. 10.13: Average `nftables` firewall throughput for rule sets generated by the different transformation algorithms, for input rule sets with two actions. The throughput for an empty firewall is shown for reference purposes.

priori reduction vs. original), 5.7× (RuleBender without a priori reduction vs. FC), and 5.8× (RuleBender with a priori reduction vs. FC).

Interestingly, we observe that RuleBender with a priori reduction leads to better throughput results than RuleBender without this technique, despite the fact that a priori reduction leads to longer path lengths, as we saw earlier. In order to explain this phenomenon, we perform a final experiment, where we generate synthetic rule sets in the same sizes as before, with traces that force worst-case behaviour in the sense that every packet needs to traverse the entire rule set before it is finally accepted. These rule sets are generated in two flavours: in the first variant, the rule set does not define any jumps actions, and therefore the rules are traversed strictly linearly without jumps. In the second variant, each rule defines a jump to the next rule, such that every every packet "jumps" $n-1$ times before it is accepted. For each size in $\{64, 256, 1{,}024, 4{,}096\}$, two such rule sets are generated (one with jumps and one without), with a corresponding trace of 20,000 equal 64-byte-sized packets. Using these rule sets, we perform throughput experiments as before, and repeat each measurement ten times for each of the firewall engines under scrutiny. The results of these measurements are given in Figure 10.16, which yields an explanation to the above observed effect: for each rule set size and for every firewall engine, it can be seen that the firewall throughput significantly drops when using the jump-based rule sets, in comparison with the non-jump rule sets.

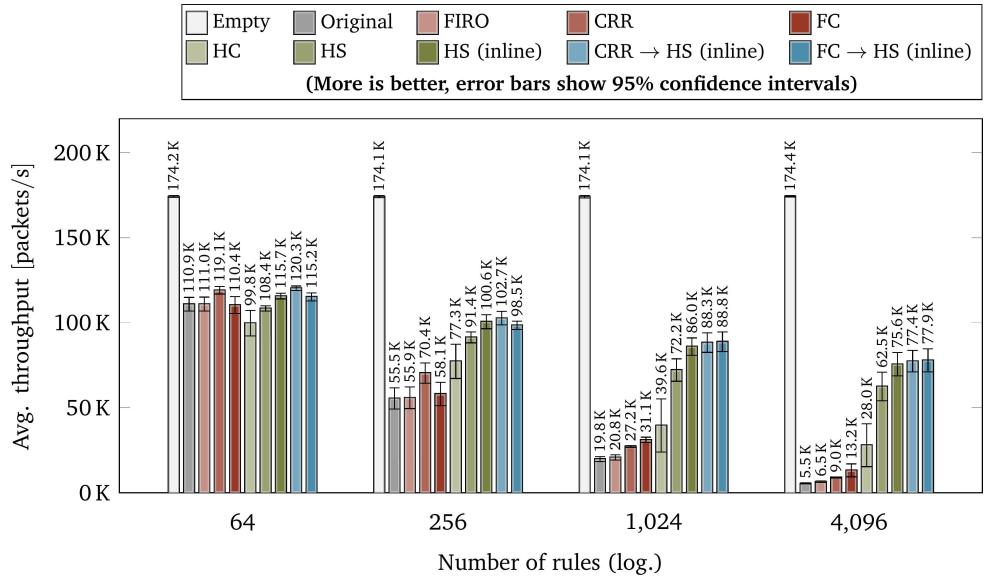


Fig. 10.14: Average `ipfw` firewall throughput for rule sets generated by the different transformation algorithms, for input rule sets with two actions. The throughput for an empty firewall is shown for reference purposes.

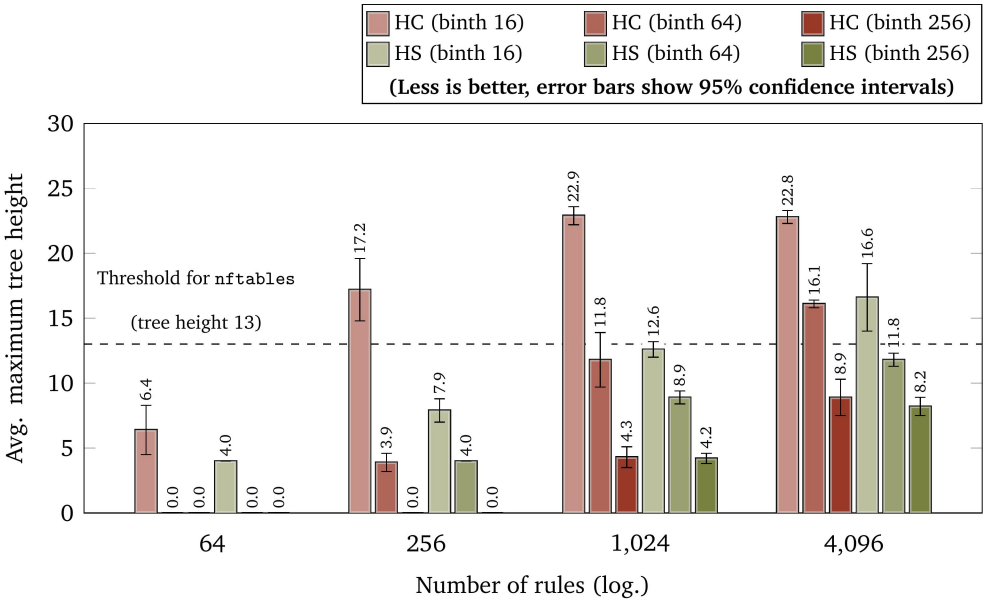


Fig. 10.15: Average maximum tree height for RuleBender with HiCuts and HyperSplit for rule sets with two actions, depending on the binth value β.

This leads us to the conclusion that the execution of jumps are more costly in terms of runtime than simple switches to the next rule. Of course, this experiment is designed to exhibit worst-case behaviour, as the RuleBender-generated rule sets execute far less jumps for the rule sets under consideration, as confirmed by Figure 10.15. Nevertheless, due to the fact that a priori reduction leads to smaller decision trees, the firewall engines need to execute fewer jumps for rule sets generated by RuleBender with a priori reduction than for rule sets generated

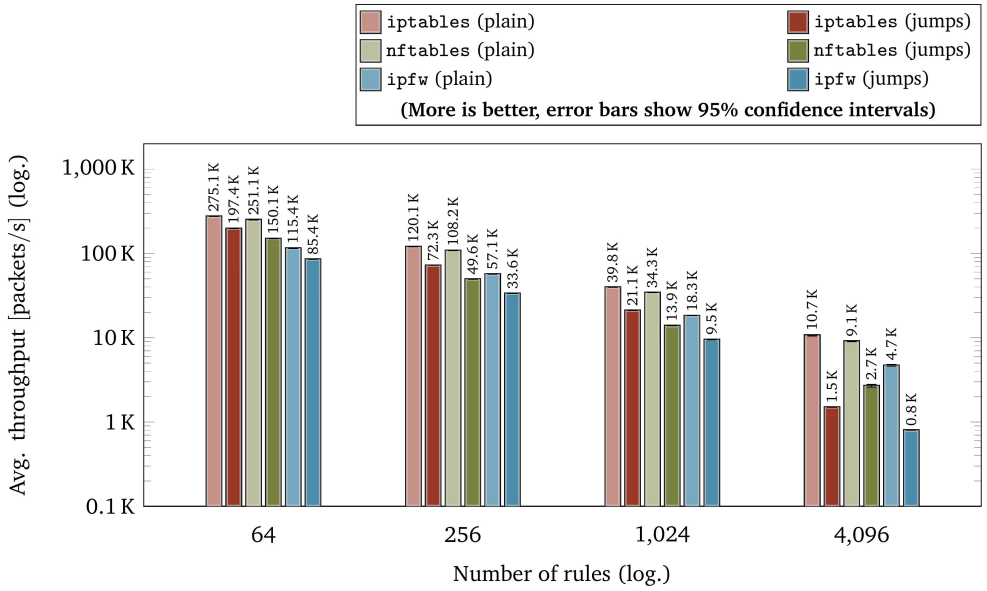


Fig. 10.16: Average firewall throughput for rule set/trace combinations either without jumps or with as many jumps as rules. The error bars are in many cases too small to be visible.

by RuleBender without this technique. In turn, the throughputs for the former rule sets are higher than for the latter ones.

10.6 Limitations

As demonstrated in the previous section, the RuleBender approach is able to significantly increase the classification throughput in the case of `iptables`, `nftables`, and `ipfw` firewall systems. However, the section also indicated that certain requirements must be met in order to (a) generate rule sets that can be loaded into the underlying classification engine, and (b) to allow the underlying engine to take advantage of the encoded decision trees.

In the case of `nftables`, we must ensure that the number of concatenated path lengths does not exceed the hardcoded threshold of 15, otherwise the generated rule set cannot be used. As another example for a system with similar constraints, we address *OpenFlow switches* [97], which organize the rule set in so called *flow tables*. Although OpenFlow switches implement jumps and can organize sub rule sets into different tables, OpenFlow versions up to `1.5.0` support only up to 254 different flow tables [174]. This is a more severe restriction than the one imposed by `nftables`, because it imposes a hard limit on the number of sub rule sets and, in consequence, on the number of nodes in the decision trees. As such, while RuleBender can theoretically be applied to OpenFlow rule sets, the binth parameter may have to be set to large values in order to restrain the decision tree

sizes, which can have a negative impact on the achievable gain in throughput (for OpenFlow systems that actually rely on a linear search for at least some flow tables).

Furthermore, the implementation of jumps can be another practical obstacle for the successful application of RuleBender. While the systems we evaluated in Section 10.5 provide jumps with reasonably good performance, this is not always the case. For example, the BSD firewall `ipf` [175] also provides jump actions in the form of "`skip` n", which ignores the next n rules and continues evaluation at the $n+1$st rule after the current rule. Unfortunately, such a skip is not implemented as a `goto` instruction, but instead actually traverses the rule set until the desired rule is reached, simply without evaluating the intermediate rules. As a result, the execution of jumps, especially when long-ranged, is basically nothing else than a simplified linear search and does not reduce the classification path length, which renders the application of RuleBender pointless for `ipf`.

Therefore, from a practical point of view, RuleBender should only be applied to rule sets for a classification system S, if S fulfills the above requirements (a) and (b). Otherwise, we advise to use an existing reduction-based scheme, as discussed in Chapter 9.

Finally, RuleBender does not provide a strictly semantics-preserving rule set transformation if the complex checks in the input rule set are not match-based. For example, consider a complex check that uses a random number generator to probabilistically accept or drop packets. Whenever this rule is traversed, it will change the classification-relevant state due to state changes within the random number generator. Now, when considering a rule set that consists of several of these rules, RuleBender will potentially separate these rules into different tree nodes. Therefore, the RuleBender-generated rule set will likely yield other state transitions than the original rule set, because in the RuleBender case, a different number of rules is traversed.

11 Summary

In this chapter we introduced the RuleBender algorithm, a technique that transforms input firewall rule sets into larger output rule sets which encode fast decision tree search structures. To this end, RuleBender exploits *jump* actions, that redirect the matching flow of the firewall to another point in the installed rule set. That way, it is possible to encode the dispatch logic of a decision tree algorithm into the rule set itself, without the need to change the underlying classification engine implementation, which often implements a straightforward, but slow, linear search over the rule set. By specializing rule sets on the jump capabilities of the underlying system, we can significantly reduce the average classification path length of incoming network packets, which, in turn, leads to higher achievable throughputs. That way, RuleBender can significantly increase the matching performance of sequential classification systems with support for reasonably fast jumps. Hence, the RuleBender approach can be used as an optimizer for both standalone software systems as well as hybrid systems based on offloading engines that partially rely on software-based classification [25, 3, 5, 6, 18].

Next to the basic RuleBender approach, which we prototyped using the well-known *HiCuts* [63] and *HyperSplit* [107] algorithms, we presented two enhancements in order to mitigate RuleBender's main drawback: the size expansion of the generated output rule sets. The first enhancement, *branch inlining*, is a direct improvement of the decision tree encoding in the generated rule set and reduces about half the generated jump rules. The second technique, *a priori reduction*, applies an existing reduction-based transformation algorithm to generate an intermediate rule set, which is fed to RuleBender. Although this approach requires more preprocessing effort, the resulting rule sets are significantly smaller, when compared to plain RuleBender-generated rule set, and often even smaller than the original input rule sets.

When compared to related work, RuleBender provides the advantages that it can deal with nearly arbitrary complex matching criteria, in contrast to related work [71, 84, 88], which can only handle range-based checks. Also, we presented a technique to handle such complex rule sets with RuleBender, when a priori reduction is applied. Moreover, the performance of RuleBender-generated rule

sets does not suffer when the number of different actions in the source rule set increases, albeit the effectiveness of a priori reduction decreases in such cases.

Our evaluation, which is mainly based on the *ClassBench* benchmark tool [132], demonstrated that classification path length reductions of two orders of magnitude for rule sets up to $4{,}096$ rules are possible. These path length reductions, in turn, lead to significant throughput increases up to factors of 7x, 14x, and 49x for the `iptables`, `ipfw`, and `nftables` firewall engines, respectively.

Part III

FPGA-based Packet Classification

12 Introduction

Most existing packet classification systems rely on classical *software-based classification techniques*, as introduced in Part I and Part II: that is, incoming network packets are received by a network interface controller and enqueued into a *receive* or *backlog* buffer [117] typically located in the host system's DDR RAM. Subsequently, packets are fetched from the backlog buffer by the CPU, which either polls the buffer or is triggered by interrupts [117]. From this point on, a packet starts its long journey through the operating system's network stack, where various functions, such as parsing, packet classification, and egress processing must be executed. Many of these procedures, e. g., packet classification or forwarding, require several additional memory accesses, sometimes even into external RAM in case of cache misses, which further prolongates the packet processing [112]. Accordingly, tight throughput or latency guarantees can hardly be given on such systems: for example, if the installed packet classification rule set grows, the performance of *all* existing classification approaches, as discussed in Part I, will generally begin to deteriorate, either due to large utilized search data structures that exceed cache capacities or due to search times larger than $\mathcal{O}(1)$. As such, reaching throughputs beyond 10 Gbit/s is hardly feasible for existing packet classification systems based on today's general purpose CPUs [69], even on multi-core architectures with parallelizable classification algorithms [110].

Thus, in order to achieve line rates of 40 Gbit/s or higher, as may be required in data centers [143], core carrier networks [29], or heavy-load VPN endpoints [137], specialized hardware architectures are required that are able to provide a large amount of true parallelism to process many packets at the same time [46]. Due to their vast parallel computing capabilities, *Field-programmable Gate Ar-*

The notion of multi-dimensional tailor-made matching circuits, as presented in this chapter, was published as first author in [14] and [15], and was also presented on domestic conferences [16] and workshops [17]. The staged priority encoder presented in [15] was suggested by Klaus Reinhardt. An application of this concept to longest prefix matchers, alongside an architecture for partial reconfiguration and prior rule set optimization, was published as first author in [9] and [8], respectively. It was first prototyped the bachelor's thesis [19] of Daniel Bendyk, which was supervised by the author.

The described specialized matching circuitry served as the hardware matching part in Andreas Fießler's HyPaFilter [3, Section 4.1] and HyPaFilter+ [6, Section IV] systems. Furthermore, the addition of multiple pipeline stages to the generated matching circuits was described in [6, Section VIII (I)]. Finally, the combination of specialized and generic matching circuits, as well as the usage of negation vectors, was prototyped by the author and evaluated by Andreas Fießler in a joint work, which was published as co-author in [2].

rays (FPGAs) [162] and *Application-specific Integrated Circuits (ASICs)* [159] are suitable candidates for low-latency high-performance packet classification systems: FPGAs can execute thousands of logical or arithmetic operations in parallel [37, 4, 56, 15, 68, 69, 109], as opposed to the sequential processing model of conventional CPUs. In contrast to ASICs, FPGAs can be reconfigured arbitrarily often to implement different user-specified circuitry, which is a useful and valuable feature in many application domains, such as network packet processing [46, 54], database query optimization [152], image processing [35], and software-defined radio [95]. This is an inherent advantage over conventional ASICs, as it creates the possibility for extending the functionality without the need to either physically manufacture a new circuit or to rely on slow additional software processing.

Accordingly, many sophisticated FPGA-based packet processing schemes have been proposed by the research community [56, 68, 69, 109] which all rely on massively parallel and pipelined classification routines. However, although these approaches differ greatly in detail, their architectures are mostly based on a strict separation between a generic packet processing algorithm and a corresponding configuration memory that stores the employed packet classification rule set. As a result, these architectures have to process two input parameters at runtime: (1) the network packets, and (2) the in-memory data structure that represents the currently active packet processing policy.

In this part, we exploit the reconfiguration ability of FPGAs to construct low-latency classification engines in a way fundamentally different to existing approaches based on generic matching circuits: instead of implementing a conventional approach based on configuration memories on top of the FPGA, we *generate matching engines that are tailor-made for one specific rule set* $\mathfrak{R}$. That way, we eliminate the need to interpret the rule set search data structure at runtime by partially evaluating the matching circuit with respect to the rule set $\mathfrak{R}$. Hence, instead of utilizing a generic configurable circuit C, we replace all configurable variables by constants based on $\mathfrak{R}$, which yields a new circuit $C_{\mathfrak{R}}$ whose operation is fixed on the rule set $\mathfrak{R}$, as sketched in Figure 12.1. In comparison to a generic matching circuit C that is configured to implement the rule set $\mathfrak{R}$, the rule-set-specialized circuit $C_{\mathfrak{R}}$ has a significantly smaller hardware resource footprint, because it implements only the exact amount of logic to represent the semantics of the specific rule set $\mathfrak{R}$.

The implication of this approach, however, is that changes to the rule set $\mathfrak{R}$ require a rebuild of the matching circuit, which is typically significantly more time-consuming than updating the configuration memories of a generic matching circuit. Therefore, we go one step further and aim to combine the advantages of both specialized and generic matchers in a hybrid classification system. This

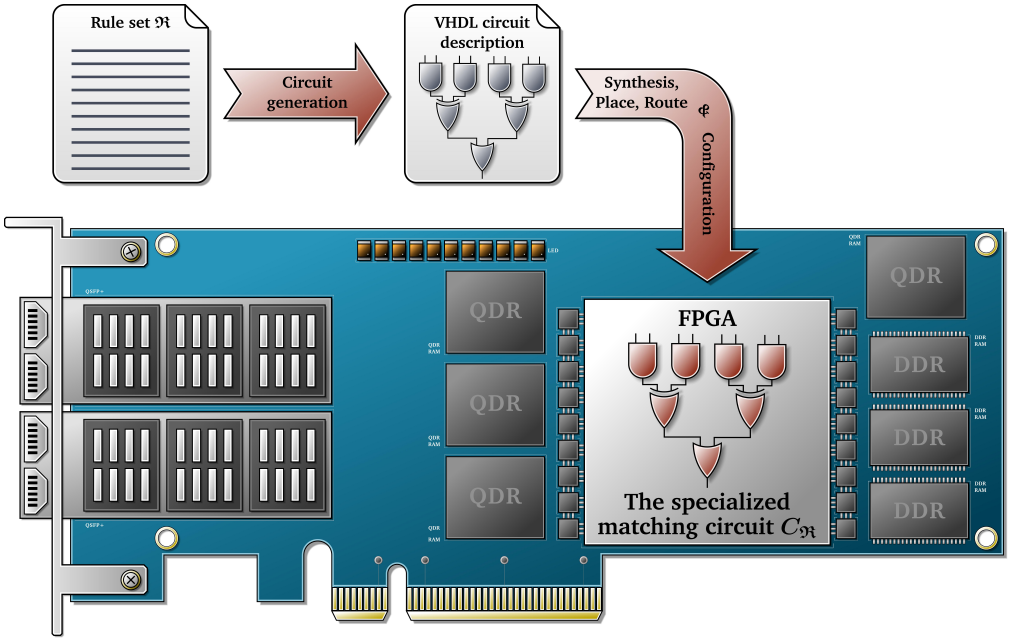


Fig. 12.1: Design flow of the Massively Parallel Firewall Circuits (MPFC) approach. A rule set $\mathfrak{R}$ is first compiled into a circuit $C_{\mathfrak{R}}$, which is subsequently programmed onto the FPGA in order to process network packets. (Parts of this figure were created by the author during his regular work at genua GmbH and are used in this work with genua's permission.)

dual-matcher pipeline is able to benefit from heavily optimized partially evaluated circuits, that are used to implement static rules, as well as the quick update capabilities of runtime-configurable matchers.

In general, the classification approaches presented in this part are designed to efficiently solve the *Geometric Packet Classification Problem*, and thus only take geometric rule sets into account. However, these techniques can also used as classification subsystems in hardware-software co-designs, which address the *Complex Packet Classification Problem* [3, 6].

The remainder of this part is structured as follows: in Chapter 13, we briefly introduce the most important aspects of FPGA architecture and design flow. Subsequently, we review related work in the domain of FPGA-based packet classification in Chapter 14. Thereafter, we describe and evaluate our proposed rule-set-specialized filtering circuits and hybrid classification pipeline in Chapter 15 and Chapter 16, respectively. Finally, we review and conclude this part in Chapter 17.

13 FPGA Fundamentals

Before we dive into the details of the proposed FPGA-based classification systems, we first pave the ground by introducing the principles of FPGA-based packet processing in this chapter. As FPGAs work fundamentally different to commonly used CPUs, we also take a brief look at the hardware architecture of common FPGAs in order to introduce key terms used in the remainder of this part.

13.1 FPGA-based Packet Classification

Traditional software-based packet classification on CPUs is mostly based on the sequential execution of several instructions in order to process incoming packets. Although conventional CPUs also provide support for true parallel computation via *instruction pipelining* [160, Chapter 12], *SIMD (Single Instruction Multiple Data) instructions* [160, Chapter 18], or *multithreaded execution* [161, Chapter 2], they are limited by the still relatively small number of CPU cores and SIMD registers, the instruction pipeline's depth, as well as undesired effects such as pipeline flushes.

In contrast, most FPGA-based classification approaches rely on custom computation units, which typically execute many operations in parallel [56, 68, 80, 109]. Furthermore, these computation units are often arranged in deep pipelines that enable the time-shifted parallel processing of multiple packets as well as the execution of hundreds or thousands of logical operations per pipeline stage. This aspect, in combination with the fact that each unit within the pipeline typically requires a single clock cycle to process its input data, in turn leads to a deterministic throughput of one classified packet per clock cycle [56, 68, 109]. Depending on the number of packets that can be injected into the pipeline per clock cycle, the system's throughput can be further increased by implementing multiple processing pipelines in parallel. Hence, despite the fact that the clock frequency, at which a larger FPGA design can be operated (typically a few hundred MHz [156, Chapter 1]), is much smaller when compared to conventional CPUs (several GHz), the achievable throughput is generally significantly higher due to the FPGA's emission of one or several classification results per cycle. The general shape of such a pipelined classification system is sketched in Figure 13.1.

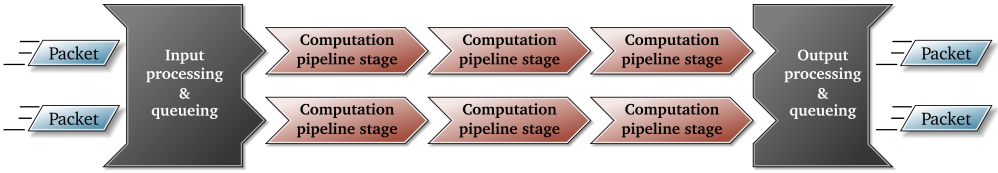


Fig. 13.1: Sketch of a general FPGA-based packet classification system consisting of two parallel processing pipelines.

However, despite the massive parallel computing capabilities provided by FPGAs, the creation of an FPGA-based classification system does not come without its own challenges. The main question that has to be answered is: given a packet rate of X packets per second and a rule set $\mathfrak{R}$ of size $|\mathfrak{R}|$, is it possible to build a classification pipeline that implements $\mathfrak{R}$ using the finite resources provided by the FPGA while processing the packet rate X, as introduced in the next section?

13.2 FPGA Architecture and Components

In order to be capable to implement custom user-defined logic functions and processing pipelines, current FPGAs consist of a multitude of small programmable logic blocks. Depending on the FPGA vendor, these logic blocks are referred to as *Configurable Logic Blocks (CLBs)* (for Xilinx FPGAs) [156, Chapter 1] or *Adaptive Logic Modules (ALMs)* (for Intel FPGAs) [181]. In the remainder of this work, we will use the Xilinx CLB nomenclature, as our experiments are based on Xilinx hard- and software. Although differing in implementation details between different vendors, the functionality of these logic blocks is similar, in that each block is capable of implementing a small number of Boolean functions. Furthermore, each CLB provides a small amount of flip-flops which can be used to store the outputs of the logic functions realized within the CLB. Hence, CLBs can be used to implement combinational and sequential logic circuitry.

To provide means for re-programmable combinational logic, CLBs contain one or more *Lookup Tables (LUTs)* [181, 42]. A lookup table is a small circuit consisting of multiplexers and SRAM cells that can store the truth tables of Boolean functions with at most k variables, with $k \leq 8$ in many of today's FPGA architectures [181, 42]. Since the LUTs' contents can be modified at will, it is possible to change the application implemented by the FPGA (nearly) arbitrarily often—which is the origin of the term "field-programmable". The composition of a three-bit LUT is illustrated in Figure 13.2.

Most non-trivial applications (or *designs*, in FPGA terms), however, cannot be implemented within a single CLB, as their combinational logic requires more than

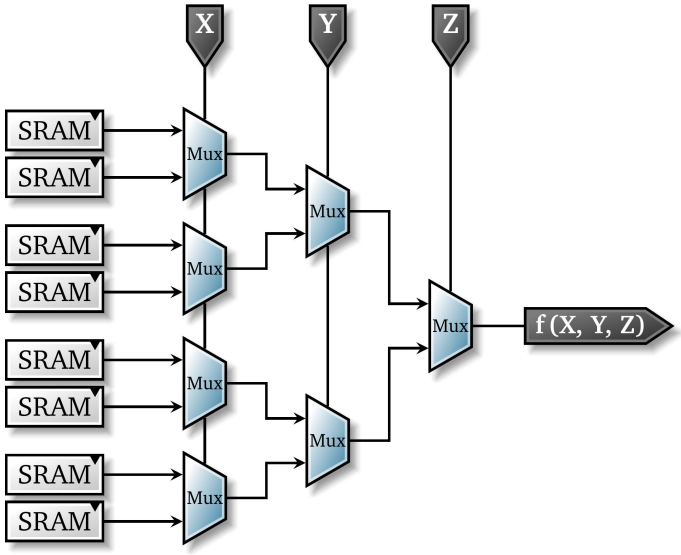


Fig. 13.2: Sketch of a three bit lookup table that can implement any Boolean function $f : \{0,1\}^3 \to \{0,1\}$ [181].

k input variables or due to more than one required pipeline stages. Therefore, the applications' logic must be subdivided into a set of smaller circuits that each can be implemented by one CLB, which are in turn logically wired together in order to implement the desired functionality. These connections are realized via a *switch matrix*, a network of busses spread throughput the FPGA's logic fabric [156, Chapter 1]. Using *Programmable Interconnect (PI)* elements, the in- and outputs of various CLBs can be linked in order to implement larger combinational or sequential logic functions. The interface to external IO components, such as network interfaces, video peripherals, or thermal sensors, is provided via *IO Blocks (IOBs)* or dedicated transceivers, through which the FPGA can communicate to external components. Finally, many FPGA vendors embed hard blocks of fixed and often required functionality in the logic fabric, such as *Block RAMs (BRAMs)*, *Digital Signal Processors (DSPs)*, or entire CPUs [157, Chapter 2]. Although some of these components can also be implemented using CLBs, it is far more efficient to provide them as fixed ASIC blocks [133]. Figure 13.3 depicts a simplified view of an FPGA logic fabric with IOBs and BRAM hard blocks.

In principle, an FPGA is able to implement any logic function that is small enough to be fit within the entirety of CLBs available within the logic fabric. However, in practice, more constraints have to be taken into account. One of the most fundamental criteria for an FPGA design is the *critical path length* [98], which refers to the longest signal propagation time within the design (or more specifically, within a certain clock domain of the design). If the signal propagation delay is too great, a design may not function properly at the targeted operating frequency, although it technically fits within the logic fabric. In fact, a design's

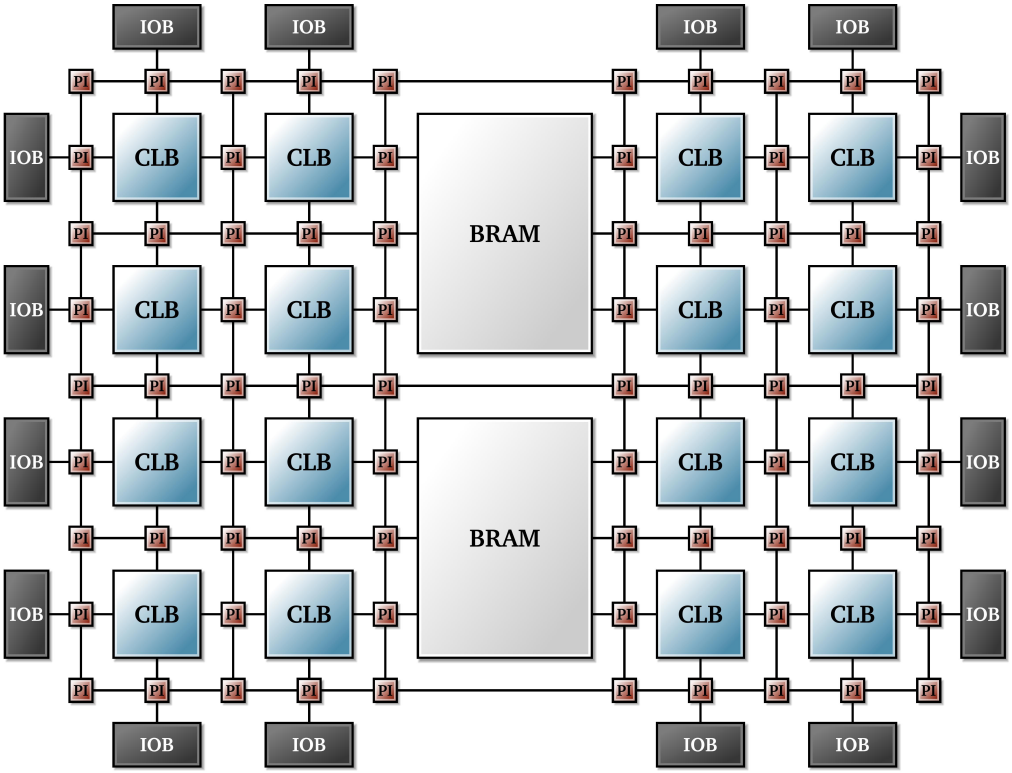


Fig. 13.3: Basic architecture of common FPGAs, showing CLBs, IOBs, PIs, and BRAMs [156, Chapter 1][157, Chapter 2][158, Chapter 11].

timing requirements is often the reason behind a deeply pipelined architecture, in order to keep the individual combinational logic blocks small [6, 56, 109]. Other constraints include heat generation, power dissipation, or the design's total computation latency.

13.3 FPGA Design Flow

In order to implement a desired functionality on an FPGA, the FPGA has to be configured using a *configuration bitstream* [153, Chapter 2]. Similar to software executables targeting a certain CPU, an FPGA design has to be compiled into a bitstream using several compilation steps, as depicted in Figure 13.4. In most cases, the input to the compilation flow is specified via the design's *Register-transfer Level (RTL)* representation in a *Hardware Description Language (HDL)*, together with a description that links the design's in- and outputs to FPGA-specific pins.

The compilation flow begins with the *analysis*, *synthesis* and *technology mapping* of the provided RTL design [153, Chapter 2]. In these steps, the design's sources are parsed and the described logic is optimized and transformed into a *netlist*, which is a network of programmable LUT primitives [48]. Thereafter, the generated

Fig. 13.4: The typical FPGA compilation flow.

netlist is *placed* onto the actual physical programmable cells (i. e., the CLBs) of the targeted FPGA. Accordingly, the FPGA's interconnect network is used to implement the connections between the individual CLBs and other components, which is referred to as the *routing* step. Finally, the synthesized, mapped, placed, and routed design is translated into the configuration bitstream.

Although the FPGA compilation flow is conceptually similar to the compilation of software programs, it is significantly more time-consuming. This is due to the fact that many of the executed compilation steps face computationally hard problems. For example, the synthesis step performs a logic optimization, which is known to be NP-hard [48, 140]. Likewise, the technology mapping and routing steps cannot be solved efficiently [48, 91, 140]. As these problems are generally not solvable in a reasonable amount of time, CAD tools rely on approximation algorithms and heuristics such as the *Espresso* logic minimizer [154, Chapter 4] in order to assemble non-optimal configuration bitstreams. Although the generated FPGA circuit configuration is not optimal, e. g., due to non-minimal logic circuitry, it still provides a significantly higher quality than the unoptimized RTL design input.

In the proposed MPFC approach, we inherently exploit the FPGA compilation flow's optimization steps, as the rule sets are embedded into the RTL design *before* it is optimized. Hence, MPFC matchers are considerable more efficient in terms of logic utilization and power dissipation than approaches that utilize generic matching circuitry which load rule sets at run time.

14 Related Work

Packet classification based on FPGAs and TCAMs has been an active topic of research since the early 2000s. Hence, we review existing FPGA/TCAM-based classification techniques in this section and point out the differences to the presented Consul approach. Generally, most of these techniques rely on generic, i. e., configuration-memory-controlled, matching pipelines, which stand in sharp contrast to the specialized MPFC matching circuits.

14.1 Ternary Content-addressable Memory

We begin our survey of existing hardware-based classification schemes by reviewing the most widely deployed matching circuit type, namely *Content-addressable Memory (CAM)* and *Ternary Content-addressable Memory (TCAM)* [102]. Although CAM circuits are not a matching technique designed or especially well suited for FPGA implementation [37], ASIC-based CAMs are still the de-facto standard used for fast packet classification [30, 67, 120, 126]. Furthermore, CAMs serve as a baseline benchmark for many FPGA-based matching approaches [56, 68, 69, 109], and also inspired a series of hybrid matching algorithms [125, 136] as well as numerous Static RAM (SRAM)-based CAM emulation approaches for FPGA implementation [27, 67, 108, 134, 135, 150].

As the name suggests, a CAM circuit is addressed via the contents of a search word, in contrast to conventional *RAM*, which is accessed using numerical addresses. In essence, a CAM can be thought of as a "hardware hashmap" that implements key-value lookups, resolves collisions with a priority encoder and can process lookup operations in a small deterministic time. In fact, typical CAMs use pipelined architectures and provide one lookup result per clock cycle, with a constant read latency of one clock cycle. In order to deliver high lookup rates at small read latencies, a CAM with a storage capacity of k w-bit key-value-pairs (to which we refer as a *k-w-CAM*) consists of three basic components arranged in a pipeline [102]: a w-bit *key matcher* with k *match lines*, a k-bit *priority encoder*, and a *value storage* of capacity k.

The matching techniques described in this chapter exclusively refer to existing related work by other authors. These techniques are depicted in the author's words to provide an overview and understanding of existing state-of-the-art, against which the author's work can be compared.

The CAM matcher is used to compare a w-bit search word x with every stored key $k_i (1 \leq i \leq k)$ in parallel in a single clock cycle, which results in a *match vector* V consisting of k bits. Here, bit i in the vector V is set if x equals k_i, and is unset otherwise, i. e.,

$$V[i] = \begin{cases} 1 & \text{if } x = k_i \\ 0 & \text{otherwise.} \end{cases}$$

We call the circuitry which computes the ith element in V the ith *match line* L_i. Note that the match vector yields a multi-match result, since it indicates every matching stored key. Although such multi-matches are useful in certain scenarios, such as multi-match packet classification [148], we review CAMs in the context of best-match packet classification, and thus need to extract the match index i^* of the most highly prioritized set bit in V. Accordingly, in the next step, a k-bit priority encoder is applied in order to compute the $\lceil \log_2(k) \rceil$-bit index i^* which indicates the first matching key k_{i^*}. Depending on the implementation of the priority encoder, the index computation requires between one and $\lceil \log(k) \rceil$ clock cycles, as explained in Section 15.2. Finally, the index i^* is used to look up the desired value $m = v_{i^*}$ from the value storage, which is typically implemented as a simple *SRAM* access [93, 102], which requires another clock cycle. In order to illustrate the CAM lookup, Figure 14.1 sketches the described matching pipeline.

The CAM matching circuitry described so far supports only one native match operation, namely exact matches. However, as one of the primary intended use

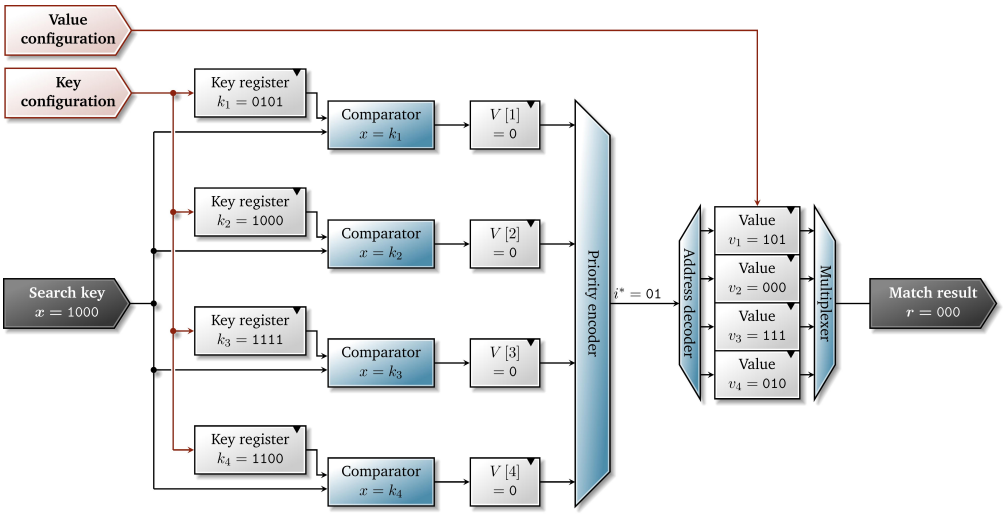


Fig. 14.1: Sketch of a 4-4-CAM circuit alongside a priority encoder and the value storage with an example configuration. Configuration busses are shown in red. In this example, searching the key $x = 1000$ yields the match vector $V = [0100]$. Accordingly, the corresponding match index is $i^* = 01$, and the match result is set to $r = 000$.

cases of CAMs is the implementation of fast longest prefix matchers [93], the restriction on only exact matches is impractical in the sense that it prevents an efficient representation of IP subnets. While it is theoretically possible to expand every subnet in a routing table and store the individual IP addresses in the CAM's match lines, this approach would require prohibitively large amounts of memory. Therefore, CAMs used in the context of packet routing and classification make use of an additional *mask operation* [93]: an operation, that was actually considered for TCAM usage much before the introduction of CIDR [180] in order to process "similarity queries" in applications [66]. The idea behind masking is to exclude certain bits of the search key x in a match line before the equality comparison is executed. To this end, each match line L_i is augmented with a *mask register* m_i, which allows the query

$$(x \wedge m_i) = (k_i \wedge m_i) \tag{14.1}$$

to be executed by each match line L_i. Each bit in the search key x that is masked by a zero bit is referred to as a "don't care" bit, and therefore, CAMs that support masked equality checks are also referred to as *Ternary CAMs*, due to the three possible states of stored bits: zero, one, and don't care. Again, we refer to a TCAM with k match lines of w bits each as *k-w-TCAM*.

Using TCAMs, it is straightforward to implement packet classification with rule sets in prefix form, because every rule directly corresponds to a match line in the TCAM, while the corresponding action codes are stored in the value storage.

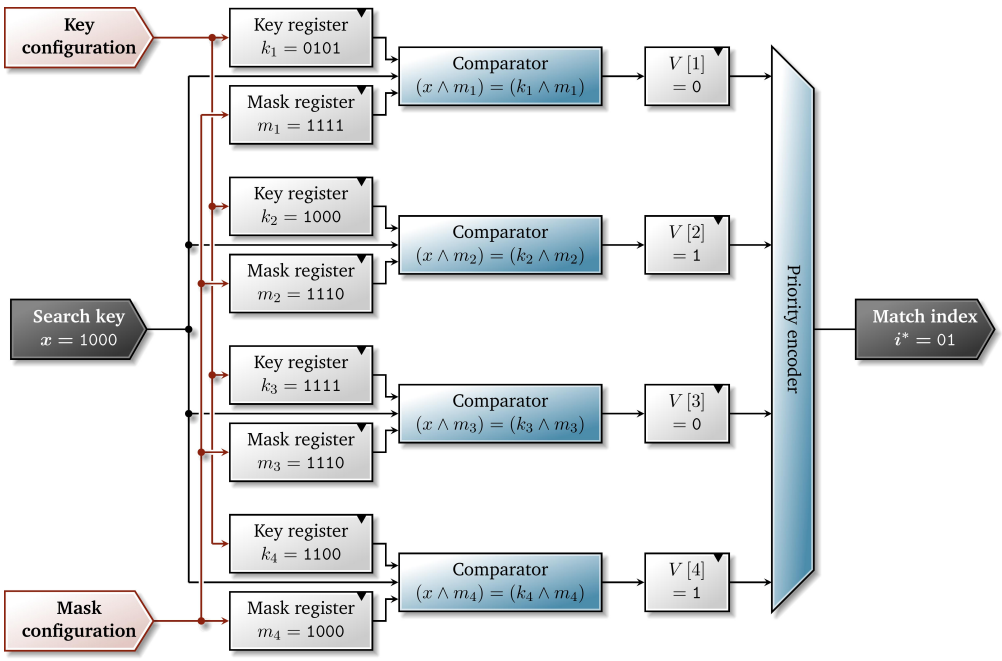


Fig. 14.2: A 4-4-TCAM with a priority encoder, alongside an example configuration. Configuration busses are shown in red.

Latency in cycles	Configuration flip-flops	Combinational operations	Supports range checks	Supports negated checks
1	$2 \cdot k \cdot w$	$2 \cdot k$ w-bit ANDs k w-bit eq. checks	no	no

k: TCAM slots w: TCAM match bit width

Tab. 14.1: k-w-TCAM performance characteristics without priority encoder.

However, problems can arise when a given rule set $\mathfrak{R}$ is not in prefix form, because some rules contain negated or range checks [43]. In this case, every non-prefix rule has to be converted into an equivalent set of prefix rules, which can lead to space explosions: every check c_j^R with w_j bits in a rule R may expand R into w_j corresponding rules [114]. The TCAM performance characteristics are summarized in Table 14.1.

In the past, the research community has proposed several approaches to mitigate the space explosion problem and to better utilize the relatively scarce available space resource in TCAMs. As a first step, it is of course possible to remove redundant rules from rule sets intended for TCAM installation, either by using general reduction techniques [49, 52, 71, 84, 88], as we have reviewed in Chapter 9, or by using reduction techniques specifically aimed at TCAM reduction [86]. Furthermore, several approaches for range encodings in TCAMs have been proposed, which aim to reduce the amount of required additional rules in case of range-to-prefix transformations [39, 40, 77]. However, neither of these approaches is able to guarantee no expansion in general, and, in some cases, require additional bits in the TCAM match rows [77]. In the case of the proposed MPFC approach, range conversions are not needed, as range checks or negated checks can be represented directly in the generated matching circuits. However, general rule set reductions *before* the circuit generation are generally useful, as we show in our evaluation in Section 15.5.

Another way of preventing space explosions due to range rule conversions is to change the layout of the TCAM: in [126], Spitznagel et al. propose a modified TCAM architecture called *Extended TCAM* that natively implements 16-bit range checks in certain area of the TCAM in order to support port range matches. They mention that this architecture requires about twice as much transistors as a standard TCAM, but this investment is rewarded by a better utilization of the individual TCAM match lines. However, this assumption only holds for rule sets that exclusively rely on range checks in the rules' port checks: if other checks also rely on ranges, such as IP ranges, as supported by various software firewalls [165, 167, 172], Extended TCAMs suffer from the same drawbacks as traditional TCAMs. In comparison to Extended TCAMs, the MPFC circuits support range checks on

every utilized check dimension, and furthermore also directly handle negated checks in the generated matcher.

When it comes to FPGA-based TCAM implementations, previous works have proposed pipelined TCAMs, whose individual match lines can be adjusted to the needs of the targeted application [96] or rule set [79, 80]. Here, the first basic idea is that the TCAM's match lines are not searched in a fully parallel manner, but in several pipeline stages in order to not break the FPGA's timing requirements. Second, in [79, 80], *irregular TCAMs* are employed, whose match lines are adjusted to the required bit width of a corresponding rule, and furthermore, whose match lines can share individual check circuits in order to preserve hardware resources. In the TCAM-branch of related research, these works are most close to our MPFC approach. The difference between MPFC and these works is that the MPFC circuits function entirely without user-controlled configuration memories, which has two important implications: first, the MPFC-generated circuitry does not utilize these memories and thus has a smaller hardware resource footprint. Second, as the rules are directly translated into rule-specialized matching circuits, we achieve an implicit optimization through logic synthesis, which allows a more fine-grained level of optimization.

14.2 StrideBV

In contrast to single clock cycle Ternary Content-addressable Memory (TCAM) matchers, as introduced in Section 14.1, the *StrideBV* technique by Ganegedara and Prasanna [56] distributes the packet classification process over multiple pipeline stages. StrideBV is specifically designed to be highly scalable with regard to the size of the utilized rule set and the number of classification-relevant header field bits. This scalability is achieved via three key design decisions in the StrideBV architecture: first, the classification process is deeply pipelined, such that each pipeline stage implements few combinational logic operations, which helps to operate the entire design at high clock frequencies. Second, StrideBV's rule set representation entirely relies on bit vectors, which can be stored in FPGA BRAMs, rather than in CLBs. Finally, the combinational logic used in StrideBV's pipeline stages is simple, because it only performs bitwise ANDs on two bit vectors. StrideBV's last pipeline stage outputs a multimatch bit vector which indicates for each rule in the installed rule set whether it matches or not. Thus, just as with (T)CAMs, a priority encoder is required to find the index of the most highly prioritized matching rule.

Before we dive into the details of the StrideBV architecture, we take a high-level look at the underlying idea of its classification operation. From a bird's eye perspective, a StrideBV matcher essentially is an SRAM-based TCAM emulation [150], which splits the incoming header bits into *strides* and computes a partial match vector in each pipeline stage. For a given *stride width* s, a rule set $\mathfrak{R}$, and the number of classification-relevant header field bits $w_{\mathfrak{R}}$, StrideBV divides the classification process into

$$k = \left\lceil \frac{w_{\mathfrak{R}}}{s} \right\rceil \tag{14.2}$$

pipeline stages p_j. Every pipeline stage p_j performs an incremental match operation that consists of a single RAM access and (except for the first stage) a vectorized AND on two bit vectors. Thereby, a partial match vector V_j of size $|\mathfrak{R}|$ is computed, which is passed to the next stage. Eventually, the final match vector V_k contains the complete match information for each rule: iff $V_k[i] = 1$, then rule R_i matches the classified packet. Finally, a priority encoder extracts the index i^* of the most highly prioritized matching rule R_{i^*} from V_k.

14.2.1 StrideBV Search Data Structure

In the remainder of this section, we assume that each rule $R_i \in \mathfrak{R}$ is given in prefix form. Hence, the rule R_i's checks can be represented as a string of $w_{\mathfrak{R}}$-many '0's, '1's, or wildcards '$*$'s, which we denote by $t(R_i)$. For example, a two-dimensional rule R in prefix form with $R = (h_1 \in 2/3, h_2 \in 8/1, A)$ over $[0, 15]^2$ has the ternary string representation $t(R) = \texttt{001*1***}$. Accordingly, the rule matches every packet p whose concatenated header bits $h_1^p h_2^p$ match the corresponding ternary bits in $t(R)$. For a given ternary string x, we refer to its *length* (i. e., the number of '0's, '1's, and '$*$'s) by $|x|$.

Given the rule set $\mathfrak{R}$ and the stride width s, the number of strides k is computed as in Equation 14.2. With k computed, the StrideBV approach divides each rule R_i into k-many *strides* $S_i^1, \ldots, S_i^k$, such that the concatenation of the S_i^js equals $t(R_i)$. For example, with $s = 3$, a rule R_i with $t(R_i) = \texttt{00101***}$ would be divided into the three strides S_i^1, S_i^2, and S_i^3 with

$$t(R_i) = \underbrace{\texttt{001}}_{S_i^1}\underbrace{\texttt{01*}}_{S_i^2}\underbrace{\texttt{**}}_{S_i^3}. \tag{14.3}$$

We refer to the widths of the individual strides by w_j, i. e., $w_j = |S_i^j|$. The first $k - 1$ strides always have a width $w_j = s$, while the width w_k can be computed as

$$w_k = \begin{cases} s, & \text{if } s \mid w_{\mathfrak{R}} \\ w_{|\mathfrak{R}|} \bmod s, & \text{if } s \nmid w_{\mathfrak{R}}. \end{cases} \tag{14.4}$$

Nr. / Priority	Field F_1	Field F_2	Ternary representation $t(R_i)$	Action
R_1	[9, 9]	[4, 7]	`1001 01**`	a^1
R_2	[1, 1]	[0,15]	`0001 ****`	a^2
R_3	[14,15]	[0, 0]	`111* 0000`	a^3
R_4	[0,15]	[10,10]	`**** 1010`	a^4

Tab. 14.2: A two-dimensional example rule set over $\mathcal{H} = [0, 15]^2$, alongside the match parts' ternary representations.

By definition, every rule R_i in the rule set $\mathfrak{R}$ is decomposed into strides of equal width, because the match parts of each rule consist of the same number of bits. Furthermore, for a given packet p, its concatenated header bits $h_1^p \dots h_d^p$ can be decomposed into *header bit strides* $S_p^1 \dots S_p^k$, such that

$$h_1^p \dots h_d^p = S_p^1 \dots S_p^k. \tag{14.5}$$

Having introduced the notion of strides, we move on to the actual StrideBV search data structure, which consists of k bit vector lists L_j. Every list L_j contains 2^{w_j} bit vectors $V_l^j (0 \leq l < 2^{w_j})$ with $w_{\mathfrak{R}}$ bits each, and each bit $V_l^j[i]$ has the semantics

$$V_l^j[i] := \begin{cases} 1, \text{ if rule } R_i's \ j\text{th stride } S_i^j \text{ matches the binary representation of } l \\ 0, \text{ otherwise.} \end{cases} \tag{14.6}$$

The lists L_j represent the contents of the RAM blocks utilized in the StrideBV classification pipeline. For any possible header bit stride S_p^j, the RAM block B_j in pipeline stage p_j stores a bit vector $V_{S_p^j}^j$ at address S_p^j. During the classification process, these bit vectors are accessed via RAM lookups, where the lookup addresses are the header bit strides S_p^j of an inspected packet p.

In order to exemplify the above described search data structure, we compute it on the basis of the rule set $\mathfrak{R}$ shown in Table 14.2 and a stride width of $s = 2$. Because each rule R_i's ternary representation $t(R_i)$ has a length of eight, four lists $L_1, \dots, L_4$ must be generated, as each pipeline stage processes two input header bits. Furthermore, each list L_j holds four bit vectors V_i^j with four bits each, since $\mathfrak{R}$ consists of four rules in total. Thus, the StrideBV pipeline needs to provide four RAM blocks in order to store the vector lists, which are shown in Figure 14.3.

Address	Bit vector
00	$V_0^1 = 0101$
01	$V_1^1 = 0001$
10	$V_2^1 = 1001$
11	$V_3^1 = 0011$

(a) RAM B_1, storing the vector list L_1.

Address	Bit vector
00	$V_0^2 = 0001$
01	$V_1^2 = 1101$
10	$V_2^2 = 0011$
11	$V_3^2 = 0011$

(b) RAM B_2, storing the vector list L_2.

Address	Bit vector
00	$V_0^3 = 0110$
01	$V_1^3 = 1100$
10	$V_2^3 = 0101$
11	$V_3^3 = 0100$

(c) RAM B_3, storing the vector list L_3.

Address	Bit vector
00	$V_0^4 = 1110$
01	$V_1^4 = 1100$
10	$V_2^4 = 1101$
11	$V_3^4 = 1100$

(d) RAM B_4, storing the vector list L_4.

Fig. 14.3: RAM blocks storing the StrideBV vector lists for the rule set in Table 14.2.

14.2.2 StrideBV Classification Pipeline

Once the vector lists L_j have been computed in a preprocessing step and written to the corresponding RAM blocks B_j, the StrideBV classification process is straightforward: when a packet p with header h^p has to be classified, its concatenated header bit strides $S_p^1 \dots S_p^k (= h^p)$ are fed into the classification pipeline. In each pipeline stage p_j, the jth header bit stride S_p^j is used as an address into the RAM block B_j to look up the partial match vector $V^j_{S_p^j}$. Intuitively, the vector $V^j_{S_p^j}$ indicates all rules R_i that match on the header bit stride S_p^j. This is similar to the classical Bit Vector algorithm where each partial vector represents matches in a single dimension, as discussed in Section 4.3. Of course, in each pipeline stage, all header bit strides that have not already been used for vector lookups must be preserved in registers r_j for the subsequent stages. Since a rule R_i only matches h^p iff it matches *all* header bit strides, these vectors are ANDed bitwise within the pipeline to compute the *result vector* V^{res}. As V^{res} indicates all matching rules, it is finally given to a priority encoder in order to compute the index i^* of the most highly prioritized matching rule R_{i^*}. This process is illustrated in Figure 14.4 for the rule set shown in Table 14.2.

Although considerably more resource-efficient than TCAMs [56], the StrideBV approach shares some of the TCAM drawbacks: first, it can only implement rule sets in prefix form, and thus may require range-to-prefix conversions, which can lead to memory explosions [56, 109]. Second, it needs to transform rule sets with negated checks, as they are also not inherently supported. When compared to

our proposed MPFC approach, StrideBV still requires significantly more hardware resources, as shown in Table 14.3, which we demonstrate in the remainder of this section.

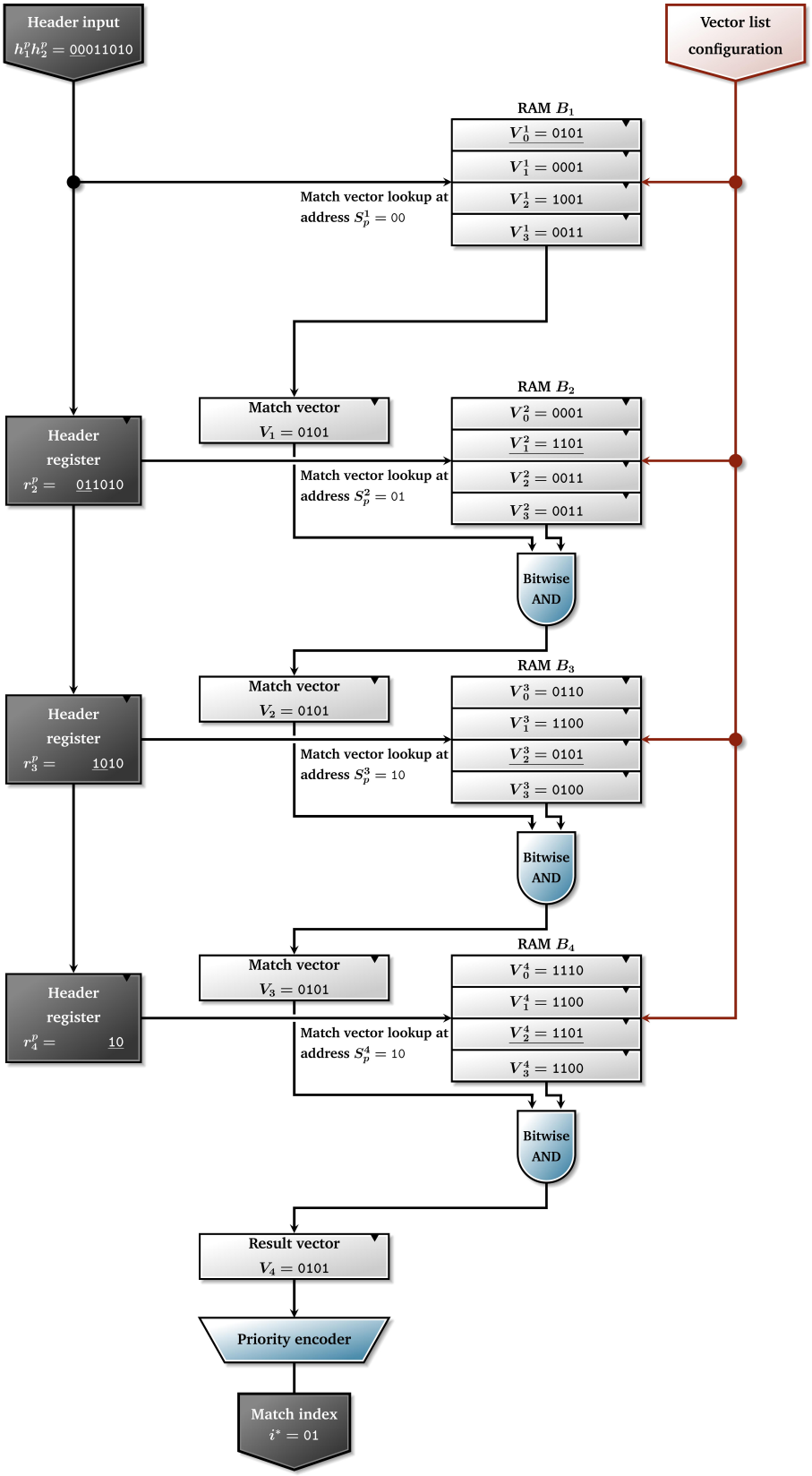


Fig. 14.4: Sketch of a StrideBV matcher with four pipeline stages. Its RAM blocks are configured with the vector lists from Figure 14.3. For illustration purposes, the classification of a packet p with the header fields $h_1^p = 0001$ and $h_2^p = 1010$ is shown.

Latency in cycles	Configuration flip-flops	Combinational operations	Supports range checks	Supports negated checks
$\mathcal{O}\left(\lceil \frac{w_{\mathfrak{R}}}{s} \rceil\right)$	$\mathcal{O}\left(2^s \cdot \lceil \frac{w_{\mathfrak{R}}}{s} \rceil \cdot n\right)$	$\mathcal{O}\left(\left(\lceil \frac{w_{\mathfrak{R}}}{s} \rceil - 1\right) \cdot n\right)$ bitwise ANDs	no	no

$w_{\mathfrak{R}}$: bits per rule in rule set $\mathfrak{R}$ s: stride width n: number of rules

Tab. 14.3: StrideBV performance characteristics (without priority encoder).

The StrideBV approach is but one of several architectures which use the striding technique to decompose the classification problem into smaller sub-problems. StrideBV's predecessor algorithm, the *Field-Split Bit Vector* approach [68], is a hybrid classification approach which relies on (T)CAMs to classify fields without ranges, such as IP addresses, and uses a bit vector approach with a stride width of one for range-based fields. In [109], a two-dimensional classification pipeline is proposed, which further decomposes the combinational logic required to AND the bit vectors into multiple sub-steps, thereby aiming to reduce the critical path length.

14.3 Further Approaches

Besides the TCAM and StrideBV approaches, as discussed in Section 14.1 and Section 14.2, respectively, many other FPGA-based classification techniques have been devised. These techniques are not necessarily limited to stateless packet classification, but may also execute other performance-critical tasks, such as Deep Packet Inspection (DPI) or TCP stream processing [46].

Sangireddy and Somani proposed an approach for FPGA-based IP packet forwarding [118] which builds upon circuitry generated from Binary Decision Diagrams (BDDs) [41]. A similar approach, also based on BDDs, was presented by Sinnapan and Hazelhurst [123]. The BDDs are programmatically generated from the forwarding table or rule set that is to be implemented on the FPGA. While conceptually closely related to our proposed MPFC technique, in that it generates rule-set-specialized circuitry, the BDD approach suffers from one central drawback: when a BDD is generated from a logic function, the BDD's size can grow exponentially with the number of input variables of said function. To make things worse, this behaviour is largely dependent on the variable ordering used within the BDD creation, and typically, good variable orderings are not necessarily known a priori [41]. Furthermore, there exist logic functions that cannot be represented efficiently using BDDs [41].

Several architectures are based on the application of hash functions in order to quickly locate matching rules. For instance, Puš and Kořenek devised a classification system which first uses a classical decomposition approach to find one-dimensional matches [106]. These one-dimensional match results are then fed into a perfect hash function to compute the actual matching rule. Due to the perfect hash function, their approach is capable of line speed execution, as the general drawback of hash-based approaches in the form of collision is avoided. However, the achievable performance may suffer from performance penalties when external RAM has to be used to store the hash table. Another example of FPGA-based packet processing with hash functions is given in [50], which proposes a DPI engine that relies on Bloom filters in order to scan packet payloads for predefined patterns. Both of these approaches do not generally rely on FPGAs as the underlying execution platform and could, e. g., also be implemented on fixed-function ASICs, in contrast to our proposed MPFC approach.

Another important challenge in high-speed network applications is the *parsing* of incoming packet headers in order to extract the header fields relevant for subsequent classification purposes. In [29], Attig and Brebner presented an approach to *generate* FPGA-based packet parsers based on declarative textual description of the parse target in a language named *PP*. The parser generator generates a synthesizable parsing pipeline based on this description, which is controlled through microcode instructions stored within the individual elements of the parsing pipeline. Hence, the generated pipeline does not necessarily have to be re-generated in order to change its behaviour. A subsequent work by Gibb et al. [57] discuss different general design principles of packet parsers for switching ASICs. They evaluated the area cost of programmable parsers, which is about twice as much as those of fixed-function parsers. Their observation confirms that the idea of specialized circuits is not tied to our MPFC approach, but can also be applied to further components of packet processing pipelines.

Generalizing the programmable switch parser idea from [57], Bosshart et al. proposed the *Programming Protocol-independent Packet Processors (P4)* switch programming language in [36]. Targeting either ASIC or software switches (e. g., OpenFlow switches), P4 is intended as configuration language for packet processing pipelines. However, as P4's primarily addresses hardware that is fixed-function in nature, it can only be used with devices or software that support P4 semantics. Although recently proposed, P4 has already been subject for case studies [70, 124] and has been used as a frontend language for both software switch [121] and FPGA [142] pipeline compilers. Naturally, using FPGAs as the target platform provides the advantage that the specified pipeline elements can be *generated* if needed. A similar approach based on a declarative language called *PX* has been introduced by Xilinx, which also compiles a high-level datapath

description into an FPGA application [38]. Prior to P4 and PX, another approach to describe the functionality of a multi-FPGA-pipeline was invented by Hadžić and Smith [64]. Their approach, which is also called *P4 (Programmable Protocol Processing Pipeline)*, is intended to specify a chain of protocol offloading engines. Both P4s and the PX approaches, as described above, are semantically orthogonal to the proposed MPFC technique: while the P4s and PX are used to *describe* entire processing pipelines, MPFC *optimizes* a single pipeline element, which, in our case, is the packet classification unit. Of course, these approaches can be combined: if one or several parts of a packet processing pipeline's configuration is known at compile time, it may be partially evaluated in order to generate more efficient circuitry.

15

Ruleset-specialized Matching Circuitry

In this chapter we present the *MPFC* approach, a technique to automatically generate highly parallel and compact matching circuitry for a specified rule set $\mathfrak{R}$. In contrast to generic matching circuits like TCAMs or StrideBV, as introduced in Chapter 14, MPFC circuits provide only the minimal required functionality to implement the semantics for the specific rule set $\mathfrak{R}$. Of course, the concept of specialized packet processing circuits requires an implementation platform whose circuitry can be adapted in case of a rule set change—a task, for which FPGAs are ideally suited for. Due to their highly compact structure, policy-specialized circuitry is a step towards closing the gap between FPGAs and fixed-function ASICs, which are, for a fixed functionality, still significantly more efficient than FPGAs [37].

Before we dive into the details of MPFC circuit generation, we motivate the general approach by considering a generic matching circuit that performs prefix matches on four-bit addresses, as shown in Figure 15.1. In this example, the registers A and M are the configuration memories, which store the net address and the subnet mask, respectively. To match an incoming address I, the circuit performs an equality check on the input address I and the net address stored in A, which are both previously masked by the contents of M. Thus, the specific configuration shown in Figure 15.1 with $A = 1011$ and $M = 1100$ tests whether the first two bits of the address are equal to 10. However, for this specific configuration, this functionality can also be realized by a single AND gate with one negated input, as shown in Figure 15.2. The specialized circuit does not only require a smaller amount of logic gates, it also does not require storage elements for the policy configuration at all.

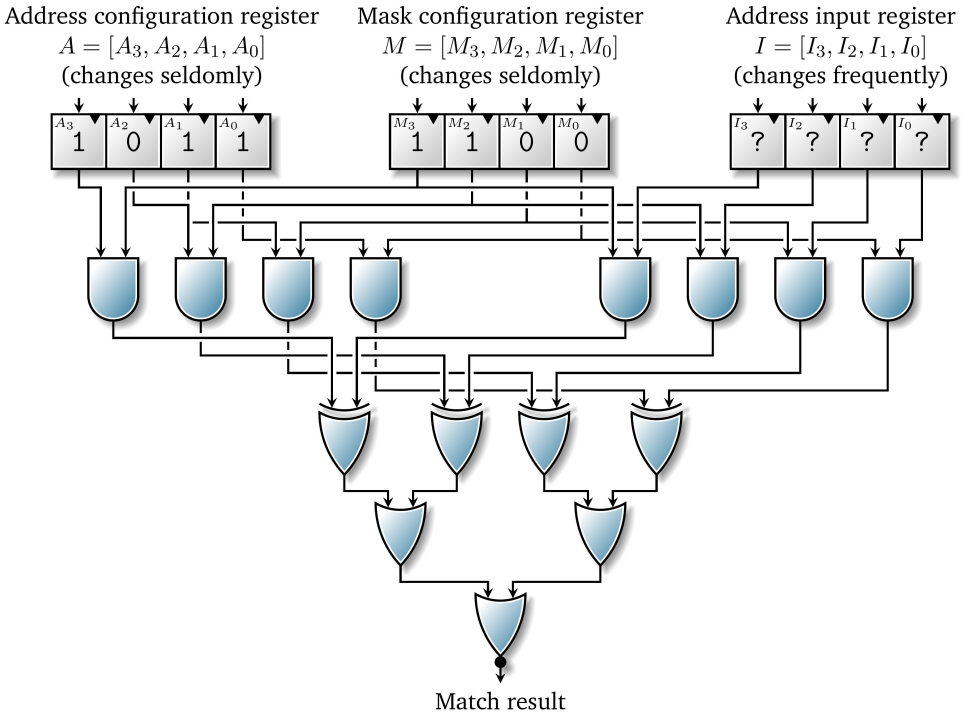


Fig. 15.1: A simple generic subnet matching circuit C for four-bit addresses that implements the Boolean function $f_C := (A \wedge M) \equiv (I \wedge M)$. The matching of an input address I is guided by the contents of the configuration registers A and M, with $A = \texttt{1011}$ and $M = \texttt{1100}$ in this example.

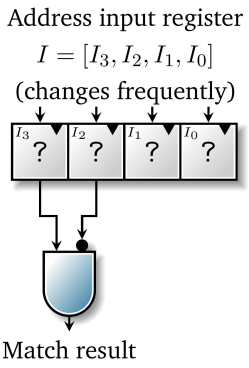


Fig. 15.2: The partially evaluated circuit $C_{M,A}$ for $A = \texttt{1011}$ and $M = \texttt{1100}$. $C_{M,A}$ implements the function $f_{C_{M,A}}(I_3, I_2, I_1, I_0) := I_3 \wedge \overline{I_2}$. Note that for all $I \in \{0,1\}^4$, $f_{C_{M,A}}(I) = f_C(I)$ when C's configuration registers are set to $A = \texttt{1011}$ and $M = \texttt{1100}$.

The advantages of this technique over generic memory-based approaches are threefold: first, due to the fact that specialized circuits incorporate the policy in their implementation, no configuration memories and thus less hardware resources are required. Second, as the specialized circuitry only needs to implement one specific rule set $\mathfrak{R}$, it is generally significantly smaller than an equivalent generic circuit for which $\mathfrak{R}$ is inherently not known at design time. Third, the

specialized circuits can be logic-optimized with regard to the structure of the rule set $\mathfrak{R}$ because $\mathfrak{R}$ is integrated into the circuitry itself. These benefits, however, come at the cost of significantly higher preprocessing times.

Our description of tailor-made matching circuits begins in Section 15.1 with their automated generation as well as their structure. Subsequently, we move on to the description of the priority encoders used in an MPFC system in Section 15.2. The combinational MPFC circuits and the priority encoder can be used to devise a pipelined MPFC approach, as depicted in Section 15.3. We analyze the key performance indicators of the proposed MPFC matcher in Section 15.4 and evaluate our approach in Section 15.5, where it is also compared with related work. Finally, we discuss the limitations of the MPFC technique in Section 15.6.

15.1 Specialized Matcher Generation

In this section we address the generation of a rule-set-specialized matching circuit $C_{\mathfrak{R}}$ for a specified $d-$dimensional rule set $\mathfrak{R}$. To ensure line speed operation, the generated matching circuit $C_{\mathfrak{R}}$ should provide the following semantics: first, for a given packet header h^p, the circuit $C_{\mathfrak{R}}$ must compute the index of the most highly prioritized matching rule R_{i^*} in $\mathfrak{R}$ after a fixed maximum number of clock cycles t. Second, assuming that a new packet can enter the packet processing pipeline every k clock cycles, the classification must happen in a pipelined fashion, i. e., every clock cycle a must provide a new classification result, provided that a corresponding packet header was fed into the circuit t cycles before. Both of these conditions are essential to guarantee line speed operation: if it is not possible to finish a packet classification after t cycles or if pipelined processing is not possible, the classification unit may introduce packet backpressure which can eventually lead to packet drops in the system's input queue. The circuits generated by the MPFC approach as well as the TCAM and StrideBV techniques allow for pipelined packet processing as well as a deterministic processing latency. In contrast to TCAMs and StrideBV, however, MPFC circuits require a significantly lower hardware resource footprint due to their rule set specialized layout, as described in the remainder of this section.

The rule set specialized matchers generated by the MPFC approach are created in a one-pass traversal over a specified rule set $\mathfrak{R}$. Every rule R_i in $\mathfrak{R}$ is translated into a corresponding *rule unit* RU_i, which is a combinational circuit that exactly encodes the matching semantics of rule R_i: RU_i maps an incoming packet p's header fields h^p to 1 iff every geometric check g_j^i of R_i is evaluated to true, and

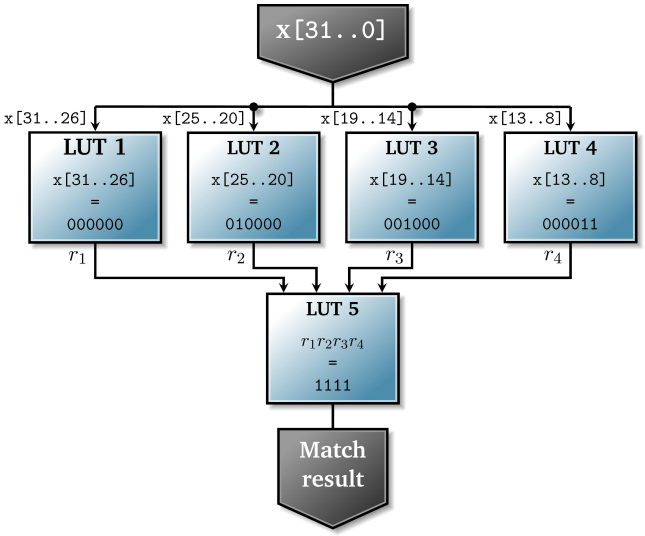


Fig. 15.3: Implementing the subnet check ($x \in 1.2.3.0/24$) in 6-LUTs. The 24 bit comparison is first distributed over four LUTs with

$$\underbrace{\texttt{x[31..26]= 000000}}_{\text{LUT 1}}\ \underbrace{\texttt{x[25..20]= 010000}}_{\text{LUT 2}}\ \underbrace{\texttt{x[19..14]= 001000}}_{\text{LUT 3}}\ \underbrace{\texttt{x[13..8]= 000011}}_{\text{LUT 4}}.$$

The final result is aggregated in LUT 5 by ANDing the outcomes of LUT 1 to 4.

otherwise to `0`. Thus, RU_i represents a partially evaluated circuit implementation of R_i's geometric match function γ_i with respect to the checks specified in R_i.

The individual header checks within a rule unit RU_i are carried out by *check units* CU_j^i, that correspond to the geometric checks g_j^i. Every check unit CU_j^i is a tailor-made combinational circuit representation of the corresponding check g_j^i and thus requires only the minimal amount of hardware resources for the implementation of g_j^i. For example, the subnet check

$$x \in 1.2.3.0/24 \tag{15.1}$$

only requires five 6-LUTs for its implementation, as sketched in Figure 15.3: while LUTs 1 to 4 execute independent six-bit comparisons, the check's result is computed in LUT 5 by ANDing the individual result bits. Just as Figure 15.2 before, Figure 15.3 illustrates the reason for the small size of partially evaluated matching circuits: the classification system's configuration parameter, i. e., the rule set, is embedded directly in the combinational logic, rather than stored in additional configuration registers which must be fed into generic matching circuitry. Also note the implicit constant folding: the input bits $x\,[7..0]$ do not have to be propagated within the subnet check and do not introduce routing overhead.

```
1  function EXTEND_CIRCUIT(Circuit C, Circuit E)
2      return new circuit by placing C in parallel to E

3  function WIRE_SOURCE_TO_SINK(Circuit C_source, Circuit C_sink)
4      return new circuit by connecting C_source's outputs to C_sink's inputs

5  function GENERATE_CHECK_UNIT(Geometric check g_j)
6      if g_j is an equality check (h_j = x) then
7          CU ← (h_j = x)
8      else if g_j is a prefix check (h_j ∈ x/y) then
9          if y = 0 then
10             CU ← (true)
11         else
12             msb ← Y_j
13             lsb ← Y_j − y
14             CU ← (h_j[msb..lsb] = x)
15     else if g_j is a range check (h_j ∈ [x, y]) then
16         if [x, y] = [0, 2^Y_j − 1] then
17             CU ← (true)
18         else
19             CU ← ((h_j ≥ x) AND (h_j ≤ y))
20     if g_j is negated then
21         CU ← not(CU)
22     return CU

23 function GENERATE_RULE_UNIT(Rule R)
24     RU ← (true)
25     for geometric check g_j ∈ R do
26         CU_j ← GENERATE_CHECK_UNIT(g_j)
27         RU ← RU AND CU_j
28     return RU

29 function GENERATE_MPFC_MATCHER(Rule set ℜ)
30     V ← generate registered match vector circuit of size |ℜ|
31     C_match ← empty circuit
32     for rule R_i ∈ ℜ do
33         RU_i ← GENERATE_RULE_UNIT(R_i)
34         RU_REG_i ← WIRE_SOURCE_TO_SINK(RU_i, V[i])
35         C_match ← EXTEND_CIRCUIT(C_match, RU_REG_i)
36     P ← generate priority encoder for |ℜ|-bit input vector
37     C_MPFC ← WIRE_SOURCE_TO_SINK(C_match, P)
38     return C_MPFC
```

Algorithm 15.1: Pseudocode that sketches how MPFC circuits are generated for a specified rule set $\mathfrak{R}$, using the entry function GENERATE_MPFC_MATCHER.

Nr. / Priority	Source subnet	Destination subnet	Source port	Destination port	Transport protocol	Action
R_1	143.17.0.0/16	8.8.8.8/32	1024-65536	53	UDP	Accept
R_2	143.17.0.0/16	8.8.8.8/32	1024-65535	53	TCP	Accept
R_3	143.17.0.0/16	*	1024-65535	80	TCP	Accept
R_4	*	*	*	*	*	Drop

Tab. 15.1: Firewall rule set to be translated with MPFC ('*' denotes a wildcard).

When each rule R_i has been translated into a rule unit RU_i, their output signals are fed into a registered match vector V that stores the individual rule match results. Subsequently, the match vector bits are used by a priority encoder in order to compute the index i^* of the most highly prioritized matching rule R_{i^*+1} (note that i^* starts at zero). Also, the priority encoder emits a *valid* signal which indicates whether at least one rule matches the current packet header. The matching index i^* is finally used to retrieve the action a^{i^*+1} from an (also generated) action Read-only Memory (ROM) to be executed onto the packet p.

Algorithm 15.1 shows the pseudocode that illustrates the above described MPFC matcher generation, with the exception of the value ROM. The function GENERATE_CHECK_UNIT is of special interest, as it describes the translation of a geometric check g_j. Depending on the type of check, it generates the tailor-made circuit representation. If g_j is a wildcard, the constant `true` circuit is emitted, which is constant-folded during the synthesis process. Furthermore, note that MPFC provides a simple and efficient way to represent negated checks by negating the outcome of the generated check unit, as shown in lines 20 and 21. In contrast, the TCAM and StrideBV approaches are not able to natively implement negated checks due to their fixed circuit structures [148]. The same holds for range checks: while MPFC generates partially evaluated range check circuits, both TCAMs and StrideBV require rule set transformations in order to represent ranges [45, 114].

We use the example rule set shown in Table 15.1 in order to visualize the structure of the corresponding generated MPFC match circuit, which is shown in Figure 15.4. The figure exhibits the parallel arrangement of the generated rule units RU_i, which contain the conjunction of the check units for the corresponding rules R_i. By taking a look at the example rule set, it is possible to identify several optimization opportunities, namely common checks between different rules and wildcard checks. The MPFC approach exploits these opportunities by generating minimal required combinational check units. For example, the check unit for the subnet test

$$\text{src} \in 143.17.0.0/16 \tag{15.2}$$

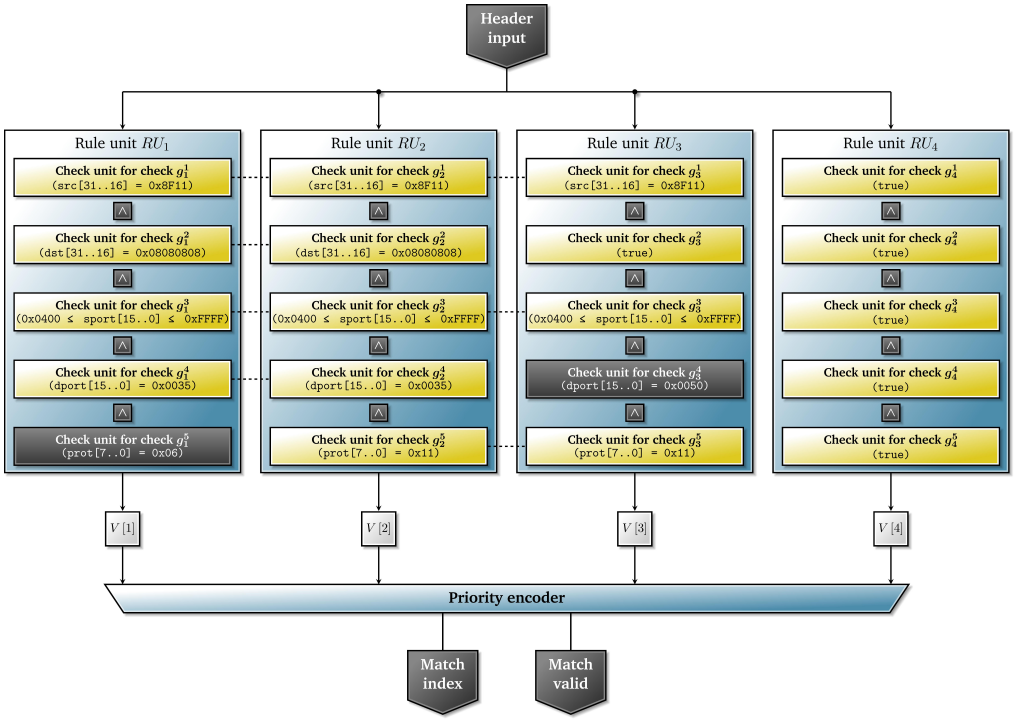


Fig. 15.4: Schematic of a combinational MPFC rule set match circuit for the rule set given in Table 15.1, connected to a registered match vector and a subsequent priority encoder. The yellow checks shown in the match units are optimized during synthesis by common subexpression elimination [47] or constant folding [145]. For example, checks that are connected with dotted lines represent common subexpressions, while checks marked as `true` are constants to be eliminated.

inspects only the most significant 16 bits of the source address input, while wildcard checks are constant-folded [145]. The elimination of common subexpressions in the form of common checks is executed during the logic synthesis phase of the circuit. Of course, it is also possible to execute this step in the MPFC generation step, but the achievable gain is questionable, because synthesis tools such as Xilinx' Vivado or Intel's Quartus perform logic optimization at a significantly more fine granular level [48]. Finally, it can be seen that range checks are directly implemented as partially evaluated comparisons. For instance, the range check

$$\text{sport} \in [1024, 65535] \tag{15.3}$$

is implemented using a single LUT that implements the function

$$\text{sport}[15] \vee \text{sport}[14] \vee \text{sport}[13] \vee \text{sport}[12] \vee \text{sport}[11] \vee \text{sport}[10]\,. \tag{15.4}$$

Using the direct range to prefix conversion method described in [45], a TCAM would require six match lines to represent a rule with this check, because the range is transformed into the six prefixes

$$\begin{aligned}
&\texttt{000001**********} && \text{(the range } [1024, 2047]) \\
&\texttt{00001**********} && \text{(the range } [2048, 4095]) \\
&\texttt{0001**********} && \text{(the range } [4096, 8191]) \\
&\texttt{001**********} && \text{(the range } [8192, 16383]) \\
&\texttt{01**********} && \text{(the range } [16384, 32767]) \\
&\texttt{1**********} && \text{(the range } [32768, 65535])\,.
\end{aligned} \tag{15.5}$$

However, although more efficient range encodings are at hand [39, 40, 77], already a single TCAM slot requires a multiple of the hardware resources used by MPFC to represent this check, as the generic TCAM always must take the full sixteen bit of the header field into account.

In summary, this section presented the first building block of the MPFC approach, namely the generation of rule set specialized parallel match circuit descriptions. In order to fully solve the Geometric Packet Classification Problem, the match vector emitted by an MPFC circuit has to be converted into the index of the most highly prioritized matching rule, or, if such a rule does not exist, into a non-match symbol. This is accomplished by a *priority encoder*, as discussed in Section 15.2.

15.2 Priority Encoder

Similar to many existing hardware-based classification systems [56, 69, 109, 125], an MPFC matcher requires a *priority encoder* circuit in order to extract the index i^* of the most highly prioritized matching rule R_{i^*} from a computed match vector V. Such a priority encoder's performance is crucial for the classification throughput, because it must be at least as fast as the classification operation in order to not bottleneck the pipeline. As such, we describe the structure of the priority encoders used within an MPFC pipeline in this section.

The problem solved by a priority encoder can be stated as follows: for a vector

$$V\left[n-1..0\right] = \left[V\left[n-1\right], \ldots, V\left[0\right]\right] \tag{15.6}$$

of n bits, we want to compute the index i^* of the leftmost set bit $V[i^*]$. Furthermore, we want to compute a *match indicator* μ which indicates whether at least one bit in V is set, i. e.,

$$\mu := \bigvee_{i=0}^{n-1} V[i]. \tag{15.7}$$

The match indicator μ tells us whether the computed index i^* contains a meaningful result. In the most straightforward design, a priority encoder can be implemented as a sequence of multiplexers and OR gates which compute the tuple (i^*, μ) in a single clock cycle. However, with increasing input vector sizes, such an approach does not scale due to the growing required amount of logic which can result in a decreasing achievable clock frequency [56].

Therefore, we employ a pipelined priority encoder in shape of a complete balanced binary tree, which is built in a recursive manner and which is well-defined for input vectors of size $2^k, k \in \mathbb{N}$. This is a valid approach, because it is always possible to left-pad the input vector with zeros, if its size is not a power of two. We assume that our approach is similar to the priority encoder of logarithmic height previously used by the authors of [56], which, however, do not provide any implementation details.

For a given $k \in \mathbb{N}$, our priority encoder is recursively defined by the function

$$\text{prio}_k : \mathbb{B}^{2^k} \to \mathbb{B}^k \times \mathbb{B}, \tag{15.8}$$

that maps input vectors of 2^k bits to the index of i^* of the leftmost set bit, represented as a k-bit integer, and the match indicator μ. The base case of the *prio* function for a two-bit vector $V[1..0] = [V[1], V[0]]$ is defined as

$$\text{prio}_1(V) := (V[1], V[1] \vee V[0]). \tag{15.9}$$

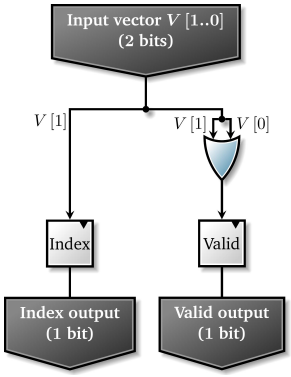
The general recursive case for a vector $V\left[2^k - 1..0\right]$ for $k > 1$ is given by

$$\text{prio}_k\left(V\left[2^k - 1..0\right]\right) := \begin{cases} \left(i^*_{\text{left}} + 2^{k-1}, \mu_{\text{left}} \vee \mu_{\text{right}}\right) & \text{, if } \mu_{\text{left}} = 1 \\ \left(i^*_{\text{right}}, \mu_{\text{left}} \vee \mu_{\text{right}}\right) & \text{, otherwise} \end{cases} \tag{15.10}$$

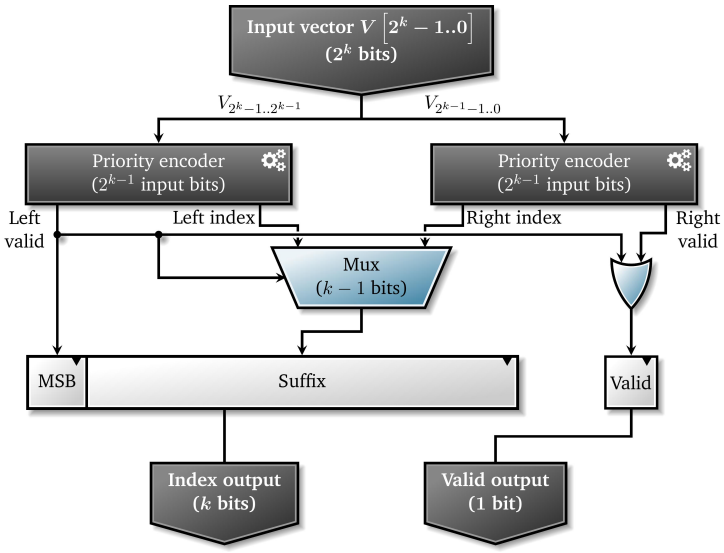
with

$$\begin{aligned} \left(i^*_{\text{left}}, \mu_{\text{left}}\right) &= \text{prio}_{k-1}\left(V\left[2^k - 1..2^{k-1}\right]\right) \\ \left(i^*_{\text{right}}, \mu_{\text{right}}\right) &= \text{prio}_{k-1}\left(V\left[2^{k-1} - 1..0\right]\right). \end{aligned} \tag{15.11}$$

The circuit representations of Equation 15.9 and Equation 15.10 are shown in Figure 15.5a and Figure 15.5b, respectively. The figures illustrate the pipelined



(a) Priority encoder for input vectors with two bits.



(b) Recursive priority encoder for input vectors with more than two bits.

Fig. 15.5: Structure of the recursive priority encoder employed in the MPFC pipeline. Each stage in the priority encoder is registered. Figure 15.5a shows the recursion's base case, while Figure 15.5b illustrates the recursive circuit.

tree shape of the priority encoder, since all intermediate results are registered independently. Furthermore, note that the addition operation in the first case of Equation 15.10 can be implemented efficiently by concatenating a single most significant bit register to the partial recursive result index, as shown in Figure 15.5b. Because every node in the priority encoder's tree shape can execute its simple operations independently in a registered manner, this circuit structure is well suited for FPGA placement and routing. This, in turn, allows the processing of large input vectors at high clock frequencies.

We point out that this priority encoder structure slightly differs from the one described in our publication [15], which is also tree-shaped, but directly yields the most highly prioritized rule R_{i^*}'s action a^{i^*} instead of the matching index i^*. The

action-based priority encoder allows for blockwise optimization and therefore has a smaller hardware resource footprint in comparison to the index-based priority encoder. However, not yielding the index of the most highly prioritized matching rule has two drawbacks: first, it hinders the collection of matching statistics on a per rule basis. Second, the MPFC matching index is required for a hybrid operation with a generic matcher, as described in Chapter 16. Therefore, we employ the index-based priority encoder in the remainder of this work.

15.3 Pipeline Structure

For a specified rule set $\mathfrak{R}$, a corresponding generated MPFC circuit computes the matching information for each rule in $\mathfrak{R}$ in parallel in a single clock cycle. As described previously, the individual match results are subsequently fed into a priority encoder to determine the most highly prioritized matching rule. While this approach allows for a pipelined operation at low processing latencies in $\mathcal{O}\left(\log\left(|\mathfrak{R}|\right)\right)$, it can be subject to diminishing returns with an increasing number of rules due to the critical path delay [98] in the rule set match circuit. Because the rule set is translated into a parallel array of rule units, the critical path length can increase with a growing number of rules, which can eventually negatively affect the achievable clock frequency. This circumstance becomes problematic when the clock frequency is too low for line speed packet processing.

In order to avoid this performance bottleneck, the MPFC approach can generate a pipeline of *partial match circuits*. Here, the idea is to partition the rule set $\mathfrak{R}$ into k partial rule sets $\mathfrak{R}_i$ by sequentially splitting $\mathfrak{R}$, i. e.,

$$\mathfrak{R} = \langle \underbrace{R_1, \ldots, R_{\left\lfloor \frac{|\mathfrak{R}|}{k} \right\rfloor}}_{\mathfrak{R}_1}, \underbrace{R_{\left\lfloor \frac{|\mathfrak{R}|}{k} \right\rfloor + 1}, \ldots, R_{2\left\lfloor \frac{|\mathfrak{R}|}{k} \right\rfloor}}_{\mathfrak{R}_2}, \underbrace{R_{2\left\lfloor \frac{|\mathfrak{R}|}{k} \right\rfloor + 1}, \ldots, R_{|\mathfrak{R}|}}_{\ldots} \rangle \tag{15.12}$$

which are independently translated into corresponding MPFC circuits C^i_{MPFC}. These circuits process incoming packet headers in a pipelined manner and are connected by *priority resolvers*. The priority resolvers are used to ensure that, once a matching rule R_{i^*} has been found in a partial match circuit C^j_{MPFC}, the index i^* is not overwritten by a subsequent match circuit C^k_{MPFC} with $k > j$. Also, a resolver ensures that the computed indices in the individual circuits are added to the correct offset, as each circuit C^j_{MPFC} only computes indices between 0 and $|\mathfrak{R}_j| - 1$. More specifically, a priority resolver PR_j placed behind a partial match circuit C^j_{MPFC} implements the function $\mathit{resolve}_{\mathfrak{R},j,k}$ with

$$\text{resolve}_{\mathfrak{R},j,k} : (\mathbb{N} \times \mathbb{B})^2 \rightarrow (\mathbb{N}, \mathbb{B}) \tag{15.13}$$

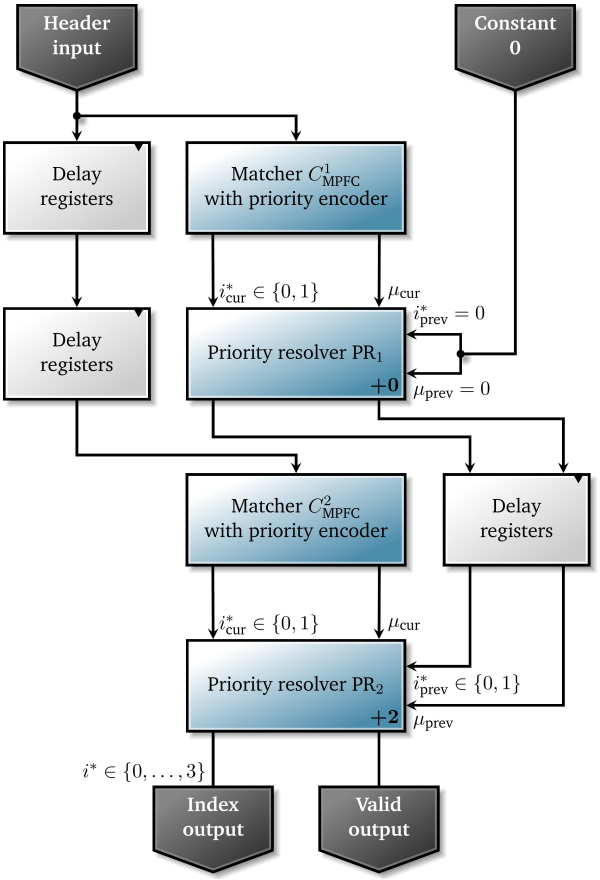


Fig. 15.6: Schematic of a pipelined MPFC matcher for a rule set with four rules, consisting of two partial matchers for two rules each.

and

$$\text{resolve}_{\mathfrak{R},j,k}\left(i^*_{\text{cur}}, \mu_{\text{cur}}, i^*_{\text{prev}}, \mu_{\text{prev}}\right) := \begin{cases} \left(i^*_{\text{cur}} + \left\lfloor \frac{|\mathfrak{R}|}{k} \right\rfloor \cdot (j-1), \mu_{\text{cur}}\right) & \text{, if } \mu_{\text{prev}} = 0 \\ \left(i^*_{\text{prev}}, 1\right) & \text{, if } \mu_{\text{prev}} = 1. \end{cases} \tag{15.14}$$

Here, the variables i^*_{cur} and μ_{cur} denote the computed match index and match valid outputs of the circuit C^j_{MPFC}, while i^*_{prev} and μ_{prev} refer to the index and valid outputs of the previous priority resolver PR_{j-1}. For the first priority resolver PR_1, i^*_{prev} and μ_{prev} are set to zero. We finish this section with an illustration of a pipelined MPFC matcher for a rule set with four rules, which is shown in Figure 15.6.

15.4 Performance Characteristics

In this section, we examine the performance characteristics of the presented MPFC approach in terms of asymptotic circuit size, processing latency, and the support for different types of checks. Just as the previously described TCAM and StrideBV

Approach	Latency in cycles	Configuration flip-flops	Combinational operations	Supports range checks	Supports negated checks
Related work					
StrideBV [56]	$\left\lceil \frac{d \cdot w}{s} \right\rceil$	$2^s \cdot \left\lceil \frac{d \cdot w}{s} \right\rceil \cdot n$	$\left(\left\lceil \frac{d \cdot w}{s} \right\rceil - 1 \right)$ n-bit ANDs	no	no
TCAM [66, 93]	1	$2 \cdot d \cdot w$	$n \cdot (d \cdot w)$-bit eq. tests $2 \cdot n$ $(d \cdot w)$-bit ANDs	no	no
Proposed approach					
MPFC	1	0	$\mathcal{O}(d \cdot n)$ $\mathcal{O}(w)$-bit eq. tests $\mathcal{O}(n)$ $\mathcal{O}(d)$-bit ANDs	yes	yes

n: number of rules d: number of fields w: bits per field s: stride width

Tab. 15.2: MPFC performance characteristics, in comparison to related work (without priority encoder).

techniques, MPFC matchers solve the Geometric Packet Classification Problem through the execution of many operations in parallel. In order to implement a d-dimensional rule set $\mathfrak{R}$ of size n with w-bit checks, the MPFC compiler assembles a circuit that consists of n parallel rule match units, each of which contains at most d check units. Thus, a total of $\mathcal{O}(n \cdot d)$ comparisons circuits for partially evaluated w-bit checks are generated. It is important to underline that this is a worst-case estimation: if the implemented rule set contains rules that define the same tests, the number of required comparison circuits shrinks accordingly due to logic optimization. Finally, if $k > 1$ pipeline stages are used for the circuit implementation, the overhead of the generated priority resolvers needs to be taken into account, which adds another $k - 1$ comparators and adders for $\log(n)$-bit operations.

Due to the fact that MPFC matchers are partially evaluated with respect to the rule set, the generated circuits do not contain any configuration registers or RAMs. Furthermore, MPFC circuits natively support both range checks and check negations and thus do not suffer from the prefix expansion problem. However, the price for the small resource footprint and check flexibility are long rule set update times, since even slight changes in the rule set require a circuit re-synthesis. The key performance indicators of the MPFC approach with one pipeline stage are summarized in Table 15.2, in comparison to related work described in Chapter 14. The table does not include resources needed for priority encoders, as these are identical for each approach.

15.5 Evaluation

In this section, we evaluate the proposed MPFC approach in terms of FPGA hardware resource utilization, power consumption, and generation time. We compare the results of the generated MPFC circuits with the two in Chapter 14 discussed generic TCAM and StrideBV matching techniques. Furthermore, we investigate the impact a translated rule set's structure has on the hardware footprint and the power consumption of a corresponding MPFC circuit. Finally, an optimality study is conducted in order to learn whether the synthesis tool's logic minimizer can be aided in the generation of smaller circuits by removing redundant rules from an input rule set *before* the MPFC translation.

15.5.1 Experiment Setup

Each MPFC data point regarding used in the evaluation in the remainder of this section is gathered using the following five steps: first, a text representation of a geometric rule set is generated, which is translated into a corresponding Very High Speed Integrated Circuit Hardware Description Language (VHDL) circuit representation in the second step. Third, the tool *Vivado 2015.1* is used to synthesize, place, and route the design, targeting a clock frequency of 180 MHz on a Virtex 7 xc7vx690tffg1761-2 FPGA. Finally, the number of LUTs and Flip-flops (FFs) as well as the power consumption estimation of the matching circuit are extracted from the log data created by Vivado after the entire build has finished. The same process is used to obtain the resource footprint and power consumption of TCAM and StrideBV matchers of a specific size, with the exception of step one. In every evaluation run, we translate only the classification part of the packet classification pipeline, without any external I/O peripherals, header parsers, or queueing modules. For every built design, we use Vivado's default implementation and optimization strategies.

In order to translate a specified rule set into a corresponding match circuit, we use the self-written C++ tool `hardbit` which was mainly developed by the author during the HARDFIRE research project. The `hardbit` tool generates a VHDL representation of complete packet classification pipeline, including the MPFC matcher, a priority encoder, as well as a value ROM for the action codes used for the treatment of classified packets. Next to the MPFC matchers, `hardbit` is also able to generate TCAM and StrideBV matchers of a specified size, against which the MPFC circuits are compared in our evaluation. Furthermore, `hardbit` also generates behavioural VHDL testbenches, which are used to verify the correctness of the MPFC, TCAM, and StrideBV matchers. Apart from the evaluation presented in this

work, the `hardbit` tool was also used to assemble the classification engines used in the hybrid classification systems *HypaFilter* [3] and *HypaFilter+* [6]. For our evaluation, `hardbit` is compiled with the `g++ 4.8.3` compiler, using the compile flags `-O0 -Wall -Werror -pedantic-errors -Wextra -std=c++0x -pthread`.

Most rule sets used in the evaluation are generated using the packet classification benchmark *ClassBench* [132], using the `acl1_seed` seed file. The only rule sets not generated by ClassBench originate from a self-written Python script which creates rule sets with a configurable amount of shared checks in each dimension, as we describe in detail in Section 15.5.3. Every generated rule set is a five-dimensional geometric rule sets, which defines checks on the source IPv4, destination IPv4, source port, destination port, and transport layer protocol fields. Due to the fact that the `hardbit` tool does only support a small fraction of all possible transport layer protocols, the transport layer protocol is encoded in a four-bit field. Therefore, the total number of classified header bits per packet is

$$\underbrace{32}_{\text{source IPv4}} + \underbrace{32}_{\text{destination IPv4}} + \underbrace{16}_{\text{source port}} + \underbrace{16}_{\text{destination port}} + \underbrace{4}_{\substack{\text{encoded transport} \\ \text{layer protocol}}} = 100. \tag{15.15}$$

For each experiment, we generated ten different rule sets for each rule set size in $\{2^i | i \in \{6, 7, 8, 9, 10\}\}$. As ClassBench does not generate actions, we uniformly distribute the two actions PASS and DROP over the generated rules, if not mentioned otherwise. All experiments are conducted on a computer running Fedora 20 Linux, which is equipped with an Intel Xeon E3-1270 v3 CPU with a clock speed of 3.50 GHz and 16 GB RAM.

The data points shown in the plots show averaged results over ten separate evaluation runs, together with the corresponding 95% confidence intervals. All data points referring to LUT usage, FF usage, BRAM usage, or power consumption are extracted from the hierarchical utilization report generated by Vivado after a successful build.

15.5.2 Hardware Resource Footprint, Power Consumption, and Build Time

In our first experiment, we compare the hardware resource footprint of the generated MPFC matchers in terms of LUT, FF, and BRAM usage as well as circuit power dissipation against the corresponding quantities of TCAM and StrideBV

matchers of equal size. We therefore translated ten different rule sets of different sizes using the `hardbit` tool and synthesized, placed, and routed the resulting design using Vivado. For each rule set size, we also generated one TCAM and one StrideBV matcher of the same size, as the structure of these generic matchers is not influenced by specific rule set layouts and features. For StrideBV, we used a stride width of ten in order to split the 100 bit header into strides of ten bit. Also, we configure StrideBV to store its search data structure in BRAMs rather than in distributed RAM.

Figures 15.7 and 15.8 show the average LUT and FF consumption of the MPFC matchers and the corresponding exact values for TCAM and StrideBV, respectively. It can be seen that the MPFC matchers require an up to 5 times smaller number of LUTs than StrideBV. When compared to a TCAM, the MPFC circuits require up to a factor of 41 fewer LUTs. This is explained by the significantly lower amounts of logic required by the MPFC matchers, as they implement only the minimum amount of logic for one specific rule set. In terms of FFs, we observe a similar pattern in Figure 15.8: an MPFC circuit requires up to one order of magnitude fewer FFs than StrideBV and up to two orders of magnitude fewer FFs than a TCAM. The reason for this circumstance is the fact that MPFC matchers only require FFs for the result vector and certain propagated metadata bits. In contrast, a TCAM needs a large amount of FFs to store the search data structure in the individual match lines, while the StrideBV matcher has to provide ten match vectors due to the stride width of ten.

Of all compared matching circuits, the StrideBV approach is the only technique which relies on BRAMs in order to store its search data structure. In contrast, the

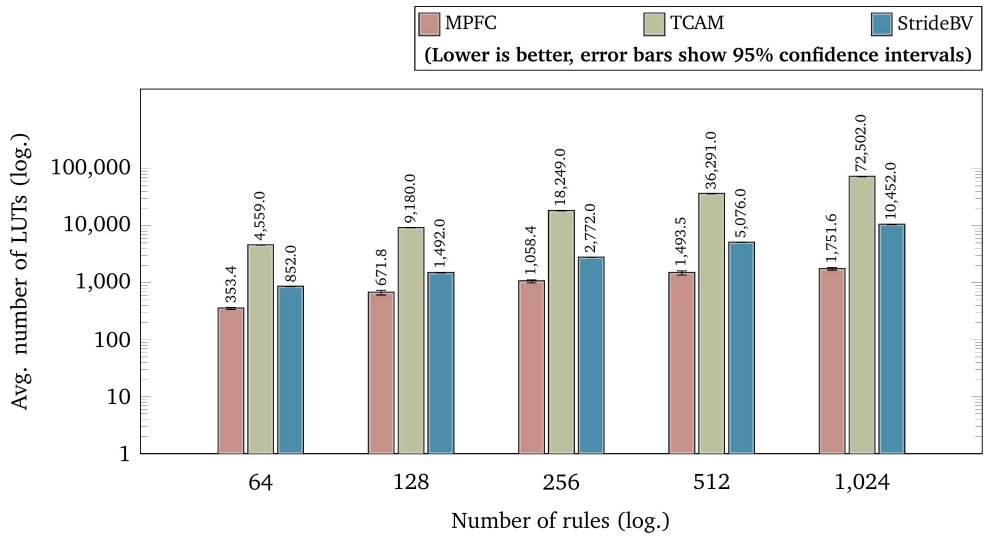


Fig. 15.7: Average LUT usage for the generated MPFC matchers, shown next to the exact LUT requirements of TCAM and StrideBV matchers of the same size.

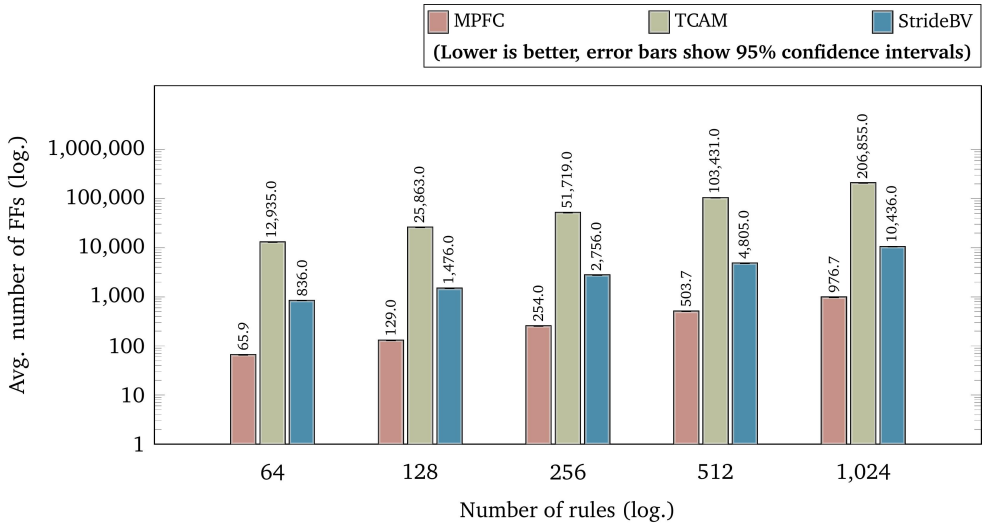


Fig. 15.8: Average FF usage for the generated MPFC matchers, shown next to the exact FF requirements of TCAM and StrideBV matchers of the same size.

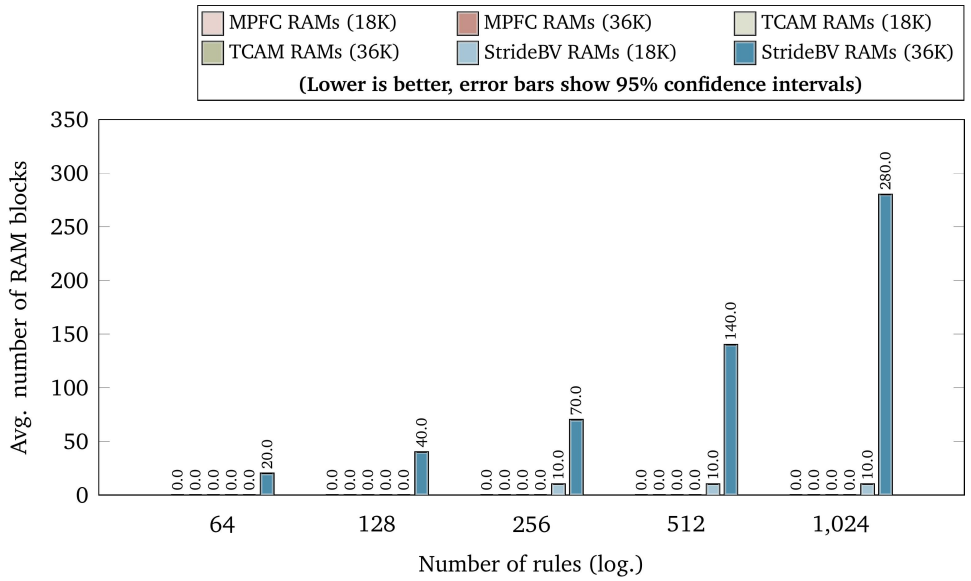


Fig. 15.9: Average block RAM usage for the generated MPFC matchers, shown next to the exact FF requirements of TCAM and StrideBV matchers of the same size.

TCAM matchers rely on distributed RAM and thus require an increased amount of FFs, while MPFC matchers implement the search data structure directly in LUT matching logic. This is confirmed by Figure 15.9, which shows the number of 18 Kbit and 36 Kbit blocks required by the individual matching circuits. For a rule set with n rules and with ten pipeline stages for each ten header bits, StrideBV needs to store $10 \cdot 2^{10}$ bit vectors with n bits each. Vivado distributes the required $10 \cdot 2^{10} \cdot n$ bits over multiple BRAMs, which can also have different capacities, as shown in the plot.

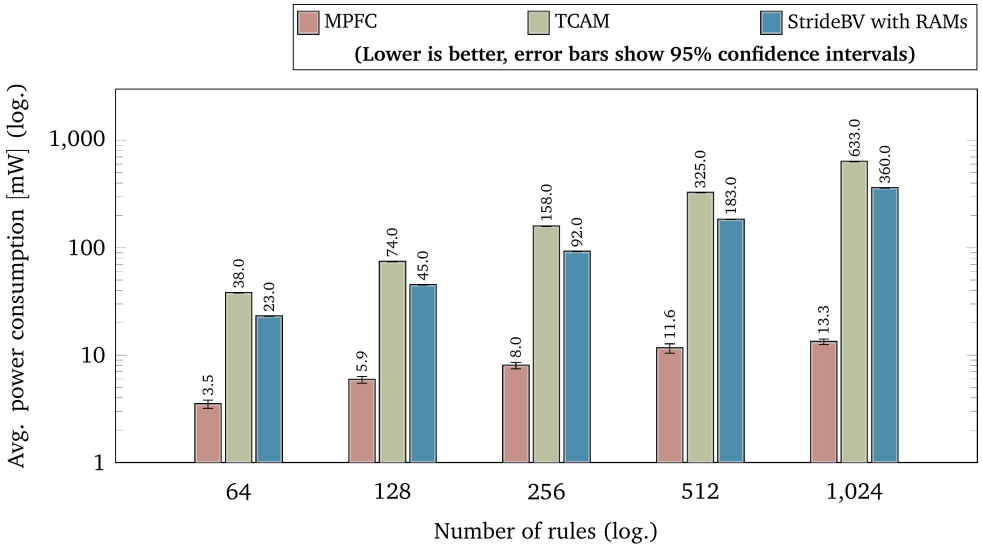


Fig. 15.10: Average power consumption for the generated MPFC matchers, shown next to the exact power consumption of TCAM and StrideBV matchers of the same size.

Figure 15.10 shows Vivado's estimation for the power consumption of the different matchers. Again, we observe that the specialized MPFC circuits dissipate about on order of magnitude less power than the generic matchers. This is expected due to the significantly smaller hardware resource footprint of the partially evaluated matchers.

Finally, we take at the look at the MPFC circuit generation and build time, which is illustrated in Figure 15.11. The plot shows that the generation of the VHDL circuit representation can be executed within few milliseconds, due to the linear time circuit construction approach. However, the plot also exhibits the main drawback of the MPFC approach: namely the required circuit implementation, which may require several minutes for the standalone matching circuitry. In a complete FPGA design with many other components, such as I/O modules or queues, building the entire design can take several hours [2]. This property clearly shows that the MPFC approach is not well suited for dynamic environments, in which the implemented rule set often changes, such as SDNs [73]. Instead, relatively static setups, such as slowly changing perimeter firewall configurations, are better suited for an MPFC-based classification system.

15.5.3 Impact of Shared Checks

In Section 15.1, we argued that MPFC matching circuits are not only smaller than generic matchers due to the missing configuration memories, but can also take advantages of shared checks within the implemented rule set: if a specific

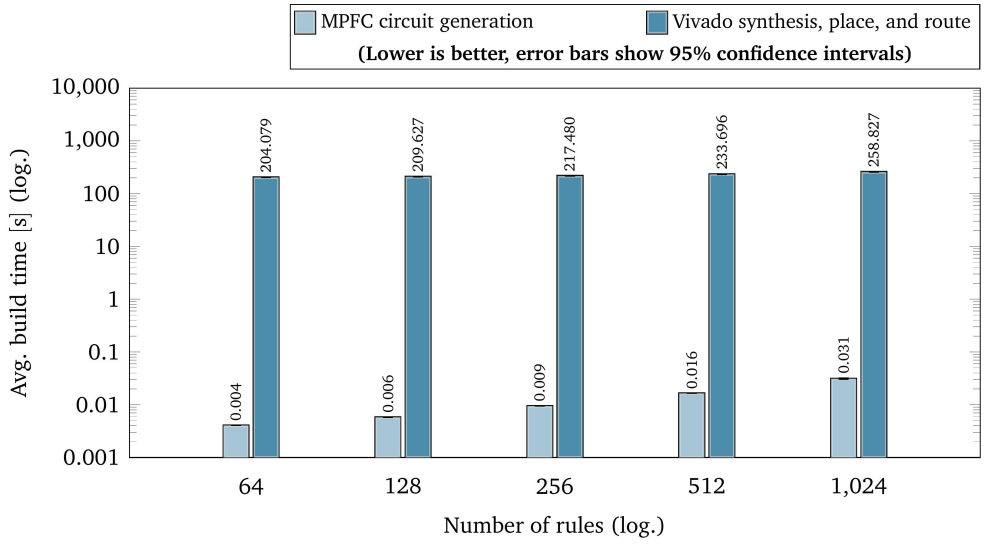


Fig. 15.11: Average build time for the generated MPFC matchers of different sizes, without priority encoder and value ROM. The entire build time is the sum of the circuit generation and Vivado times.

geometric check is specified by more than one rule, it should not have to be implemented more than once. Instead, since such checks are effectively common subexpressions [47] within the logic function upon which the specialized matcher is based, it should be possible to eliminate duplicated checks during the logic optimization step. In order to verify this assumption, we conduct an experiment in which the used rule sets specify p percent of *shared checks* and $100 - p$ percent of *unique checks* per matching dimension. To this end, we generate random rule sets of size n with a custom Python script as follows: first, for every dimension j (except for the transport layer protocol),

$$u := \begin{cases} 1 & \text{, if } p = 100 \\ \frac{n \cdot (100-p)}{100} & \text{, otherwise} \end{cases} \tag{15.16}$$

unique checks are generated. Then, $s := n - u$ *shared* checks are drawn from the set of the unique checks. These $n = s + u$ checks are randomly distributed over the jth dimension of the generated rule set. Due to the small cardinality of the set of possible values, we always randomly choose between UDP and TCP for each rule in case of the transport layer protocol dimension. Subsequently, we use the previously described build process to translate and implement the corresponding MPFC matchers.

Figure 15.12 shows the average number of used LUTs for the generated matchers. The figure reveals that LUT usage decreases with an increasing number of shared checks, which can be explained by the smaller logic functions required to implement rule sets with many shared checks. In the extreme case of 100% of shared

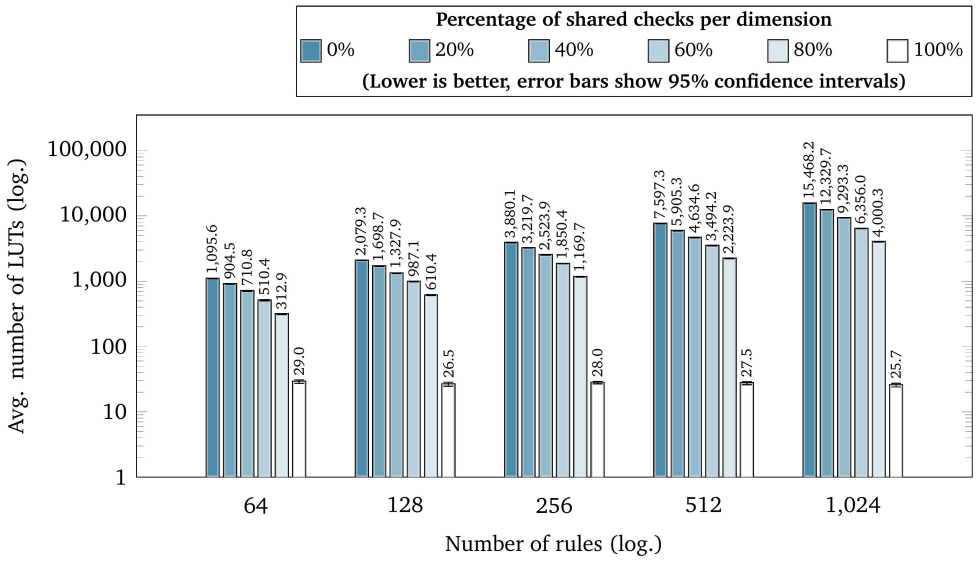


Fig. 15.12: Average LUT usage of the generated MPFC matchers, depending on the number of shared checks per dimension.

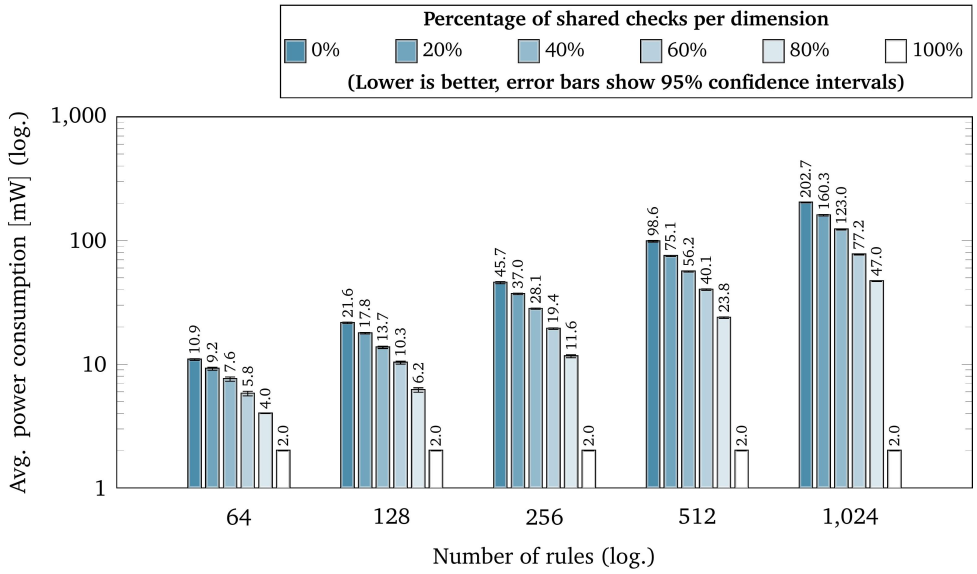


Fig. 15.13: Average power consumption for the generated MPFC matchers, depending on the number of shared checks per dimension.

checks, the rule set effectively consists of a single rule, which is the reason why, for all rule set sizes, LUT usage is nearly equal.

When comparing the results of Figure 15.12 to those shown in Figure 15.7, it can be seen that the LUT usage is significantly higher for the randomly generated rule sets. This is explained by the range tests used for source and destination port

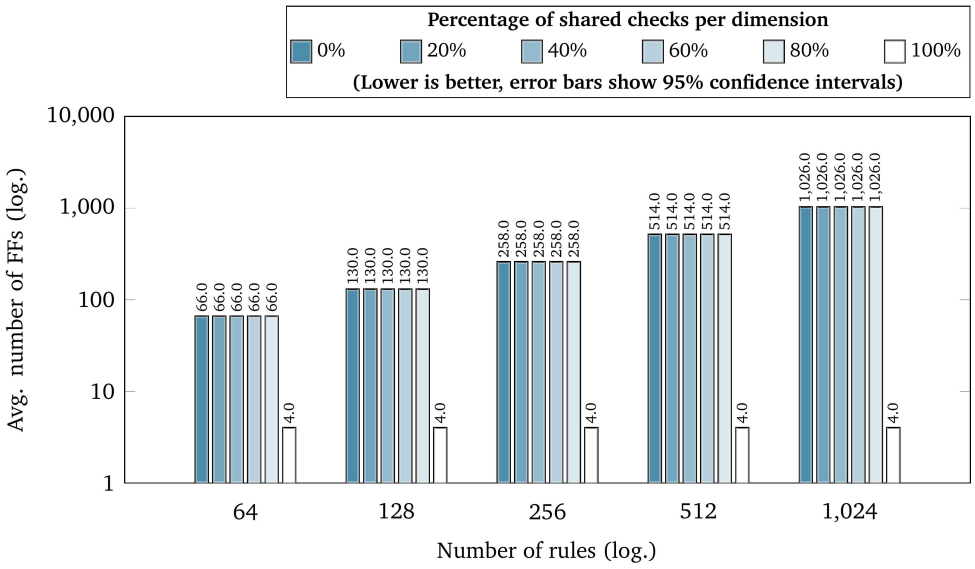


Fig. 15.14: Average FF usage of the generated MPFC matchers, depending on the number of shared checks per dimension.

ranges. In the randomly generated rule sets, we randomly choose the intervals used for each source and destination port range test from the set

$$\{(a,b)\,|a,b \in \{0,\ldots,2^{16}-1\}, b \geq a\}. \tag{15.17}$$

with equal probability. In contrast, the port ranges specified by the ClassBench rule sets are most often either wildcards or equality checks, which are simpler to implement. However, while the MPFC circuits can still natively implement these port ranges, both the TCAM and StrideBV techniques struggle with such rule sets, as they require the entire rule set to be in prefix format, which can lead to huge expansion factors [45, 114], as explained in Section 15.1.

Corresponding to the shrinking size of LUTs with an increasing number of shared checks, we can also observe a smaller power consumption in Figure 15.13. This does not surprise, since the generated circuits' power dissipation is of course dependant on the number of LUTs the circuits are comprised of.

In contrast to LUT usage and power consumption, the number of FFs does not decrease with an increasing number of shared checks, with the exception of the extreme case $p = 100$. Although the synthesis tool is able to significantly reduce the size of the combinational matcher, it still has to generate the complete match vector, as no redundant rules are detected. In the case of $p = 100$, the situation is different: here, every rule with the exception of the first rule is redundant and can therefore be eliminated.

15.5.4 Optimality Study

In our final experiment, we are interested in the optimality of the synthesized MPFC circuits with respect to the redundancy of entire rules within the specified input rule set. The previous Section 15.5.3 already shows that in the trivial case in which the entire rule set consists of copies of the first rule, the logic optimization performed during synthesis can take advantage and eliminate all rules except for the first. However, we already saw in Section 9.2 that there are more subtle possibilities for a rule to be redundant, for example if it is covered by the union of multiple more highly prioritized rules or if it is downward redundant. This raises the question whether the logic synthesis can benefit from prior rule set reduction step, which shortens the rule set *before* it is fed into the `hardbit` and Vivado tools.

To answer this question, we use the in Section 9.2 introduced CRR approach [84] by Liu and Gouda in order to remove all redundant rules from the ClassBench rule sets before they are synthesized to MPFC matchers. Subsequently, we translate the original rule sets as well as their CRR-optimized counterparts in order to examine the performance characteristics of the resulting matching circuits.

Figures 15.15, 15.16, and 15.17 show the LUT and FF usage as well as the power consumption of the generated circuits. The figures clearly reveal that the synthesis results of the original circuits is not optimal: the CRR-optimized circuits are significantly smaller in terms of LUTs and FFs and consequently consume less

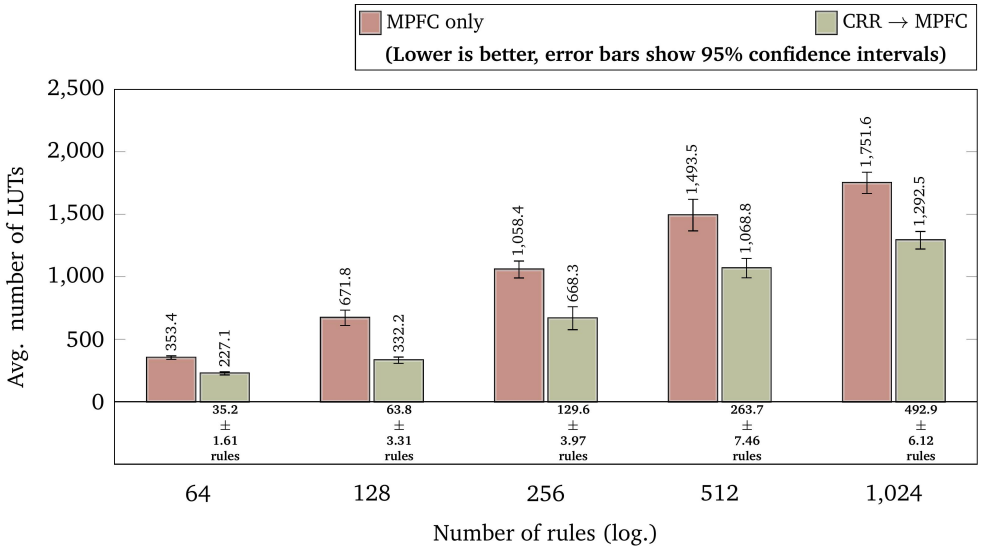


Fig. 15.15: Average LUT usage for the generated MPFC matchers for plain and CRR-optimized rule sets with two actions. The average rule set sizes (with 95% confidence intervals) used for MPFC circuit generation in case of the CRR-optimized rule sets are shown at the bar bottoms.

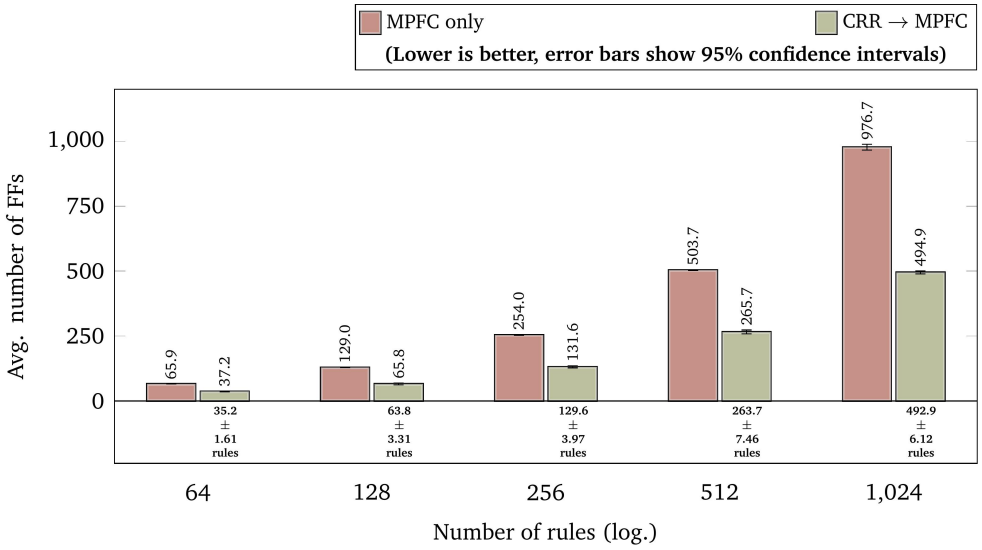


Fig. 15.16: Average FF usage of the generated MPFC matchers for plain and CRR-optimized rule sets with two actions. The average rule set sizes (with 95% confidence intervals) used for MPFC circuit generation in case of the CRR-optimized rule sets are shown at the bar bottoms.

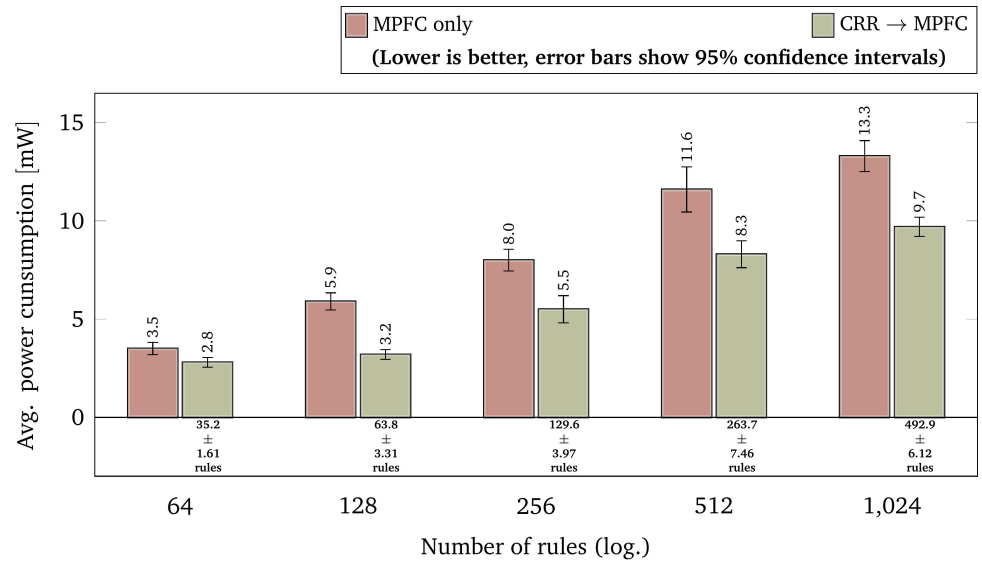


Fig. 15.17: Average power consumption of the generated MPFC matchers for plain and CRR-optimized rule sets with two actions. The average rule set sizes (with 95% confidence intervals) used for MPFC circuit generation in case of the CRR-optimized rule sets are shown at the bar bottoms.

power. From a theoretical point of view, the synthesis tool should be able to detect such redundancies, as the entire information of the rule set, including the actions, is integrated into the MPFC circuit. However, in contrast to the CRR algorithm, logic minimizers typically rely on heuristics [154] which may generate suboptimal results. While this approach is a valid approach for a general synthesis tool such as Vivado, our evaluation results demonstrate that additional optimal

domain specific techniques, such as CRR, can significantly improve the quality of the resulting circuitry.

15.6 Limitations

In comparison to generic hardware-based classification techniques such as TCAMs or StrideBV, the rule set specialized MPFC matchers provide the same throughput at significantly lower hardware resource footprint and power dissipation, as demonstrated in our evaluation. However, our experiments also exhibit that these advantages do not come without a cost, namely the time it takes to build such a circuit. While generic matchers can be quickly configured or updated during system runtime by simply re-writing their configuration memories, MPFC circuits require a costly re-implementation for every change in the implemented rule set. As such, the MPFC approach cannot be used in environments with frequent rule set changes, such as programmatically updated SDNs [73] or core routing tables [75]. Instead, MPFC is better suited for environments with infrequent changes, such as static perimeter firewalls.

Due to the fact that today's FPGAs allow circuit on-the-fly circuit replacements at runtime via *partial reconfiguration* [89], it is possible to perform rule set updates at runtime, without the need to power down the FPGA, as we demonstrated in [8, 9] at the example of partially evaluated forwarding tables. However, this still leaves us with long delays until a new MPFC circuit is available after a rule set update. Therefore, the standalone MPFC approach, as presented in this chapter, should not be considered a practical solution for typical packet filtering use cases which may require quick dynamic updates. Instead, it paves the ground for a hybrid classification system which supports incremental updates while being able to implement stable rules in a highly efficient way, which we present in Chapter 16.

16 Hybrid FPGA-based Classification

Up to this point, we have discussed two different kinds of matching circuit approaches suitable for FPGA implementation: generic techniques, on the one hand, rely on configuration memories that store the search data structure that must be fed into corresponding generic match units in order to classify an incoming packet header. Examples for these kind of approaches, such as TCAMs [66, 93] or StrideBV [56], are presented in Chapter 14. Generic matchers can quickly be configured for a specific rule set at system runtime by re-populating their configuration memories with the corresponding data structure. However, due to their generic nature, these circuits' hardware resource footprints are comparatively large, as shown in Section 15.4 and Section 15.5. On the other hand, we introduced the MPFC approach in Chapter 15, which generates specialized matching circuitry for a specified rule set. Without any runtime-configurable RAMs, the corresponding busses, and heavily optimized combinational classification logic, MPFC matchers require orders of magnitudes fewer hardware resources than their generic counterparts. However, MPFC circuits suffer from high configuration latencies, which arise every time the implemented rule set is changed even by a single bit.

With these two diametrically opposed approaches at hand, we introduce the *Hybrid Classification Circuit with Action Consolidation (Consul)* technique, which aims to combine the advantages from both worlds in a single classification pipeline. A Consul system consists of four main components: the first component is an MPFC matcher, which is used to implement stable rules that only change seldom. Second, the specialized matcher's output, alongside the packet header information, is fed into a generic matching circuit, which executes an independent packet classification and pipelines the matching result of the MPFC matcher. The generic matcher is intended to be used for quick rule insertions or modifications at runtime, which cannot be carried out by a standalone MPFC circuit. Finally, a *consolidation unit* is used to determine which of the two match results takes precedence. Moreover, we augment the MPFC matcher with a vector-based *rule negation circuit*, which allows us to dynamically disable rules embodied by the MPFC matcher.

Using these components in a pipelined fashion, Consul implements a flexible line speed matcher that can implement rules in a highly compact and optimized way as well as provide means for dynamic rule insertions, deletions, or modifications, as described in the remainder of this chapter. Moreover, Consul provides the flexibility to be configured for different use cases, as the size capacity of the two matchers can be chosen freely (of course, as long as they stay within the boundaries of the underlying FPGA implementation platform).

The remainder of this chapter is structured as follows: first, we introduce the rule set partitioning in Section 16.1, which is used to distribute a rule set over the different matching engines in the Consul pipeline, which is subsequently described in Section 16.2. Next, we focus on Consul's rule set update capabilities in Section 16.3. Thereafter, we review Consul's performance characteristics in Section 16.4 and evaluate our proposed approach in Section 16.5. Finally, we conclude this chapter by discussing Consul's limitations in Section 16.6.

16.1 Rule Set Partitioning

The main purpose of Consul's hybrid classification circuit is to provide an efficient implementation platform for two different kinds of rules, namely *static rules* and *dynamic rules*. We consider static rules to remain active and constant over a longer period of time, such that the effort of compiling a highly optimized circuit representation of them using the MPFC approach can be justified. Typical examples for static rules are static access control entries in a perimeter firewall, permanent forwarding entries, or default policies in a packet filter or router. On the other hand, dynamic rules are considered to be short-lived entries in a rule set which are only required temporarily, e. g., in order to rapidly react to threads such as Denial of Service (DOS) attacks [100]. Another use case for dynamic rules are candidates for static rules that must quickly be implemented within the classification engine until the next MPFC rebuild. Consul implements static rules using a specialized MPFC matcher M_{MPFC}, while dynamic rules are handled by a generic matcher M_{gen}, e. g., a TCAM or a StrideBV matcher, as described in Chapter 14.

As such, any given rule set $\mathfrak{R}$ that is to be implemented within a Consul classification engine must be partitioned into two sub rule sets $\mathfrak{R}^{\text{MPFC}}$ and $\mathfrak{R}^{\text{gen}}$ with

$$\mathfrak{R}^{\text{MPFC}} = \langle R_1^{\text{MPFC}}, \ldots, R_{|\mathfrak{R}^{\text{MPFC}}|}^{\text{MPFC}} \rangle \tag{16.1}$$

and

$$\mathfrak{R}^{\text{gen}} = \langle R_1^{\text{gen}}, \ldots, R_{|\mathfrak{R}^{\text{gen}}|}^{\text{gen}} \rangle. \tag{16.2}$$

For each sub rule set type T in $\{\text{MPFC}, \text{gen}\}$, we define a function

$$\pi^T : \{1, \ldots, |\mathfrak{R}^T|\} \to \{1, \ldots, |\mathfrak{R}|\} \tag{16.3}$$

that maps the index i of a rule $R_i^T \in \mathfrak{R}^T$ to the index j of the corresponding rule $R_j \in \mathfrak{R}$. Of course, each rule in $\mathfrak{R}$ is either static or dynamic, and therefore we require

$$\text{Im } \pi^{\text{MPFC}} \cap \text{Im } \pi^{\text{gen}} = \emptyset \tag{16.4}$$

and

$$\text{Im } \pi^{\text{MPFC}} \cup \text{Im } \pi^{\text{gen}} = \{1, \ldots, |\mathfrak{R}|\}. \tag{16.5}$$

Furthermore, it is important that the partitioning is stable with respect to rule prioritization in order to maintain the original rule set $\mathfrak{R}$'s matching semantics within the Consul classification pipeline. Accordingly, the relative ordering between two rules R_i^T and R_j^T in $\mathfrak{R}^T$ must be the same as the ordering of the corresponding rules $R_{\pi^T(i)}$ and $R_{\pi^T(j)}$ in $\mathfrak{R}$, and thus,

$$i < j \Rightarrow \pi^T(i) < \pi^T(j). \tag{16.6}$$

Figure 16.1 illustrates the Consul rule set partitioning as well as the functions π^T.

Having introduced the rule set partitioning scheme required by the Consul approach as well as the index mapping functions π^T, we now move onward to the circuit structure of the hybrid classification system.

16.2 Hybrid Matching Pipeline

In this section, we dive into the details of Consul's classification operation. We begin by depicting the circuit structure of the packet processing pipeline used by a Consul system, which is illustrated in Figure 16.2. The figure shows ten distinct functional pipeline components I.1 to IV.2, in addition to a packet First In - First Out Memory (FIFO) that is used to store network packets while their headers are being classified. These components are grouped into four main categories, namely packet preprocessing (I), packet classification using MPFC (II), packet classification using a generic matcher (III), and finally the packet postprocessing (IV). While packet pre- and postprocessing are common to most packet classification systems and shown only for reference purposes, the main ingredients of the Consul system are the classification stages II and III.

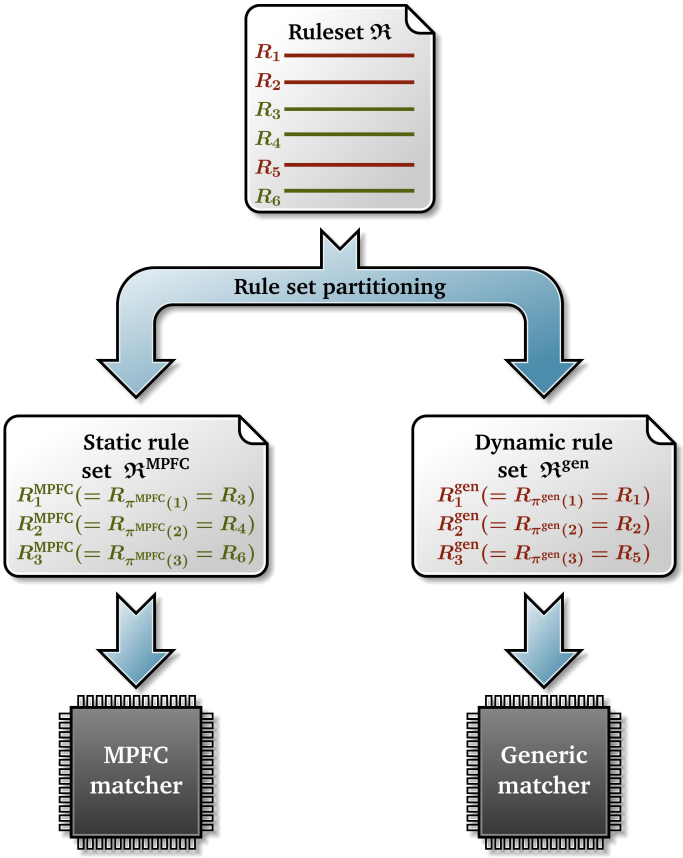


Fig. 16.1: Sketch of the rule set partitioning of the complete rule set $\mathfrak{R}$ into the sub rule sets $\mathfrak{R}^{\text{MPFC}}$ and $\mathfrak{R}^{\text{gen}}$. Static and dynamic rules are colored in green and red, respectively.

When a packet p enters the processing pipeline, the header parser (I.1) extracts the packet header h^p, which is required by the subsequent pipeline stages in order to perform the classification operations. At the same time, the packet p is enqueued into the packet FIFO, until its fate has been decided by the classification pipeline. Next, the packet header h^p is classified by the MPFC matcher M_{MPFC} in Step (II.1), as described in Chapter 15. In contrast to the plain MPFC approach, Consul adds a small runtime-programmable element, the *negation vector* (Step II.2), to the matcher, which is used to dynamically dis- and enable rules. Note that M_{MPFC} exclusively implements the rules within the sub rule set $\mathfrak{R}^{\text{MPFC}}$. We refer to the matching index computed by M_{MPFC} in Step II.3 by i^*_{MPFC}, with

$$i^*_{\text{MPFC}} \in \{1, \ldots, |\mathfrak{R}^{\text{MPFC}}|\} \cup \{i_\epsilon\}. \tag{16.7}$$

Here, the index i_ϵ stands for an invalid index, if none of the rules in $\mathfrak{R}^{\text{MPFC}}$ matches the packet header h^p. Accordingly, we refer to the corresponding action looked up by the MPFC matcher in Step II.4 by a_{MPFC}.

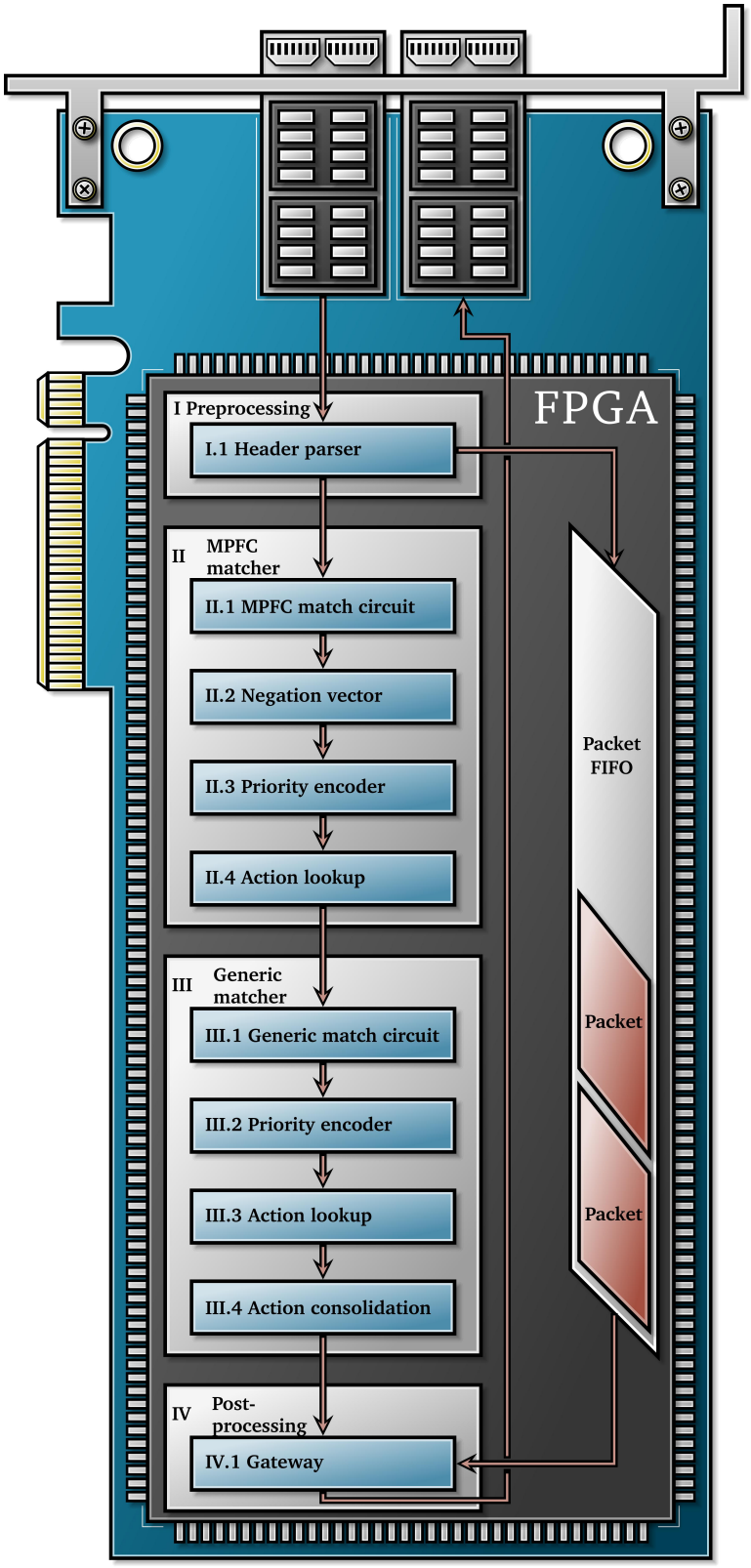


Fig. 16.2: Illustration of the packet processing pipeline used in a Consul system. The different processing steps I.1 to IV.1 are grouped into coherent blocks I to IV. The groups I and IV are common functionalities and shown for reference purposes, while the groups II and III are specific to the Consul approach.

Up to this point, the packet p has been completely classified with respect to the sub rule set $\mathfrak{R}^{\text{MPFC}}$. Due to the fact that the generic matcher M_{gen} could implement more highly prioritized matching rules within the sub rule set $\mathfrak{R}^{\text{gen}}$, a second classification must take place in Step III. This second classification is executed using a generic runtime-configurable matcher M_{gen}, which utilizes a corresponding search data structure for the sub rule set $\mathfrak{R}^{\text{gen}}$. Analogously to M_{MPFC}, M_{gen} computes the index i^*_{gen} of the most highly prioritized matching rule in $\mathfrak{R}^{\text{gen}}$ as well as the corresponding action a_{gen} in Steps III.1, III.2, and III.3.

At this point, however, the Consul pipeline must prioritize one of the two classification results produced by the two independent matchers M_{MPFC} and M_{gen}, which must happen in accordance with the semantics of the original rule set $\mathfrak{R}$. Therefore, the action lookup module in Step III.3 is not only used to look up the action a_{gen}, but also a *consolidation index* $\chi_{i^*_{\text{gen}}}$. The consolidation index $\chi_{i^*_{\text{gen}}}$ is used to decide which of the two matching results takes precedence if the situation occurs that both matchers have found a valid matching rule, i. e., $i^*_{\text{MPFC}} \neq i_\epsilon$ and $i^*_{\text{gen}} \neq i_\epsilon$. More precisely, for any rule $R_i^{\text{gen}} \in \mathfrak{R}^{\text{gen}}$, the consolidation index χ_i depicts the index of the most highly prioritized static rule $R_{\chi_i}^{\text{MPFC}} \in \mathfrak{R}^{\text{MPFC}}$ that has a lower priority than R_i^{gen} with respect to the complete rule set $\mathfrak{R}$. We describe this circumstance by defining, for a given dynamic rule R_i^{gen}, the set X_i of all indices of less highly prioritized static rules than R_i^{gen} with respect to $\mathfrak{R}$ using

$$X_i := \left\{ j \;\middle|\; R_j^{\text{MPFC}} \in \mathfrak{R}^{\text{MPFC}} \text{ with } \pi^{\text{MPFC}}(j) > \pi^{\text{gen}}(i) \right\}. \tag{16.8}$$

Note that the indices in X_i refer to the rule indices in $\mathfrak{R}^{\text{MPFC}}$.

Using X_i, we now define the *consolidation index* χ_i with

$$\chi_i := \begin{cases} \min(X_i) & \text{, if } X_i \neq \emptyset \\ \underbrace{|\mathfrak{R}^{\text{MPFC}}| + 1}_{\text{(there is no less highly prioritized static rule in } \mathfrak{R})} & \text{, otherwise.} \end{cases} \tag{16.9}$$

For each dynamic rule R_i^{gen}, we store the precomputed consolidation index χ_i alongside the rule's action $a^{\pi^{\text{gen}}(i)}$ in the action lookup module used in Step III.3.

Finally, the *action consolidation* module uses the matching indices i^*_{MPFC} and i^*_{gen} as well as the consolidation index $\chi_{i^*_{\text{gen}}}$ in order to compute a global match result a^* in Step III.4. To this end, the module needs to distinguish between four cases, as depicted in Algorithm 16.1. Three of the four cases are trivial in the sense that there is at most one match result, namely if no matcher or at most one matcher

```
1  function CONSOLIDATION(i*_MPFC, i*_gen, χ_{i*_gen}, a^{π(i*_MPFC)}, a^{π(i*_gen)})
2      if (i*_MPFC = i_ε) ∧ (i*_gen = i_ε) then
3          a* ← ε
4      else if (i*_MPFC ≠ i_ε) ∧ (i*_gen = i_ε) then
5          a* ← a^{π(i*_MPFC)}
6      else if (i*_MPFC = i_ε) ∧ (i*_gen ≠ i_ε) then
7          a* ← a^{π(i*_gen)}
8      else
9          if i*_MPFC < χ_{i*_gen} then
10             a* ← a^{π(i*_MPFC)}
11         else
12             a* ← a^{π(i*_gen)}
13     return a*
```

Algorithm 16.1: Action consolidation decision logic.

Nr. / Priority	Field F_1	Field F_2	Action	Name in $\mathfrak{R}^{\text{MPFC}}$	Name in $\mathfrak{R}^{\text{gen}}$	Consolidation index χ
R_1	$[\ 5,\ 6]$	$[\ 1, 10]$	a^1	—	R_1^{gen}	1
R_2	$[\ 2,\ 7]$	$[\ 0, 13]$	a^2	—	R_2^{gen}	1
R_3	$[14, 14]$	$[\ 0,\ 1]$	a^3	R_1^{MPFC}	—	—
R_4	$[\ 0, 15]$	$[10, 10]$	a^4	R_2^{MPFC}	—	—
R_5	$[14, 15]$	$[\ 0,\ 0]$	a^5	—	R_3^{gen}	3
R_6	$[\ 0, 15]$	$[\ 0, 15]$	a^6	R_3^{MPFC}	—	—

Tab. 16.1: A two-dimensional example rule set $\mathfrak{R}$ over $\mathcal{H} = [0, 15]^2$, partitioned into the sub rule sets $\mathfrak{R}^{\text{MPFC}}$ and $\mathfrak{R}^{\text{gen}}$ for use within Consul.

finds a valid match result. In the case that two valid match results are available, the consolidation unit checks whether

$$i^*_{\text{MPFC}} < \chi_{i^*_{\text{gen}}} \tag{16.10}$$

holds. If Condition 16.10 evaluates to true, then the rule found by M_{MPFC} must be more highly prioritized than the rule found by M_{gen}, because $\chi_{i^*_{\text{gen}}}$ refers to the index (in $\mathfrak{R}^{\text{MPFC}}$) of the most highly prioritized static rule that has a lower priority than $R^{\text{gen}}_{i^*_{\text{gen}}}$. Note that in the special case that $\chi_{i^*_{\text{gen}}} = |\mathfrak{R}^{\text{MPFC}}| + 1$, Condition 16.10 will always evaluate to true, which is correct since $R^{\text{gen}}_{i^*_{\text{gen}}}$ is less highly prioritized than any static rule. On the other hand, if Condition 16.10 does not hold, we know that rule $R^{\text{MPFC}}_{i^*_{\text{MPFC}}}$ can at most be as highly prioritized as rule $R^{\text{MPFC}}_{\chi(i^*_{\text{gen}})}$, which has a lower priority than $R^{\text{gen}}_{i^*_{\text{gen}}}$ by definition of χ.

Finally, Step IV.1 in Figure 16.2 represents the application of the computed action a^* onto the corresponding packet p in the packet FIFO. If, for example, a^* specifies

a DROP action, p will be discarded at this point, i. e., it will be read entirely from the FIFO, without forwarding it to a network transceiver. Due to the fact that the MPFC circuit can provide a new classification result with each clock cycle, the throughput of the Consul pipeline is only restricted to the throughput of the utilized generic matcher. When equipped with one of the previously described generic TCAM [93] and StrideBV [56] approaches, Consul is therefore also able to produce one classification per clock cycle.

We finish this section with a small example with for the Consul pipeline's classification operation, using the rule set $\mathfrak{R}$ shown in Table 16.1. If we consider a packet p_1 with the two-dimensional header $h^{p_1} = (14, 0)$, the matcher M_{MPFC} for the static rule set will compute $R_1^{\text{MPFC}} (= R_3)$ as the most highly prioritized matching rule, while M_{gen} will instead compute the result $R_3^{\text{gen}} (= R_5)$. As both matchers have found a valid result, the matching index $i^*_{\text{MPFC}} = 1$ is compared to the consolidation index $\chi_{i^*_{\text{gen}}} = \chi_3 = 3$. Because $1 < 3$, we know that the matching rule R_1^{MPFC} computed by M_{MPFC} is more highly prioritized than the matching rule R_3^{gen} computed by M_{gen}. In contrast, for the packet p_2 with the header $h^{p_2} = (15, 0)$, the Consul pipeline will compute the matching rules R_3^{MPFC} and R_3^{gen}, respectively. In this case, however, the comparison $i^*_{\text{MPFC}} = 3 < 3 = \chi_{i^*_{\text{gen}}} = \chi_3$ yields false, which leads to the selection of $R_3^{\text{gen}} (= R_5)$ as the most highly prioritized matching rule.

16.3 Rule Set Updates

Having discussed the Consul classification algorithm in the previous section, we now address updates on the implemented rule set $\mathfrak{R}$. Since Consul is a hybrid classification system consisting of a static MPFC matcher as well as a generic matcher with runtime update capabilities, many update operations can be executed in about the same time a purely generic classification system can be updated, because Consul update operations can be reduced to those utilized for generic matchers. However, operations that require a rebuild of the specialized MPFC circuit, still require a significant amount of time. In this section, we specifically focus on *rule insertion* and *rule deletion* operations, as they represent the minimal set of primitives in order to execute arbitrary rule set changes, when composed accordingly. For the remainder of this section, we use the rule set $\mathfrak{R} = \langle R_1, \ldots, R_n \rangle$ as our vantage point.

16.3.1 Rule Insertions

The insertion of a rule R_i, $i \in \{1, \ldots, n+1\}$, into the rule set $\mathfrak{R}$ results in a new rule set $\mathfrak{R}'$ of size $n+1$ with

$$\mathfrak{R}' = \langle R_1, \ldots, R_{i-1}, R_i, R_{i+1}, \ldots, R_n \rangle. \tag{16.11}$$

This operation is supported by Consul by marking R_i as a dynamic rule, in which case it will be added to the search data structure used by the generic matcher M_{gen}. Due to the consolidation index mechanic described in the previous section, Consul supports the full insertion index range $\{1, \ldots, n+1\}$. In case the generic matcher M_{gen} used in the Consul pipeline provides support for incremental updates, Consul rule insertions inherit this feature, because neither static nor other dynamic rules need to be adjusted. Once the updated data structure has been computed, it can be written to M_{gen}'s configuration memories, which are often implemented as register banks or RAMs [56, 68, 93, 109].

Of course, dynamic rule insertions are limited by the storage capabilities of the generic matcher M_{gen}: if no further configuration space is available, at least one rule from the dynamic sub rule set $\mathfrak{R}^{\text{gen}}$ must be either deleted or moved to the static sub rule set $\mathfrak{R}^{\text{MPFC}}$, which is implemented significantly more efficiently by M_{MPFC}. Here, a possible strategy is to select highly frequented rules in $\mathfrak{R}^{\text{gen}}$ that have remained active and constant for longer time periods, in an effort to exploit the "elephants and mice" phenomenon [103]. Although such rule migrations to $\mathfrak{R}^{\text{MPFC}}$ are expensive due to the required MPFC rebuild process, this process can still be executed at runtime due by exploiting the FPGA's partial reconfiguration capabilities [8].

16.3.2 Rule Deletions

Deleting a rule R_i from the rule set $\mathfrak{R}$ is handled in one of two ways by Consul, which depends on whether R_i is a static or a dynamic rule, i. e., if $i \in \text{Im } \pi^{\text{MPFC}}$ or $i \in \text{Im } \pi^{\text{gen}}$, respectively. If R_i is a dynamic rule, then it is implemented in the search data structure stored in the configuration memories of M_{gen}, in which case it can be removed by adapting the memories' contents. Depending on the specific generic matcher, this operation may be incremental in nature, or may require a complete data structure re-computation.

On the other hand, if R_i is a static rule, it cannot be removed from M_{MPFC} as it is a fixed component of the circuitry. However, in order to still quickly carry out the deletion semantics, Consul uses a slight augmentation of the original

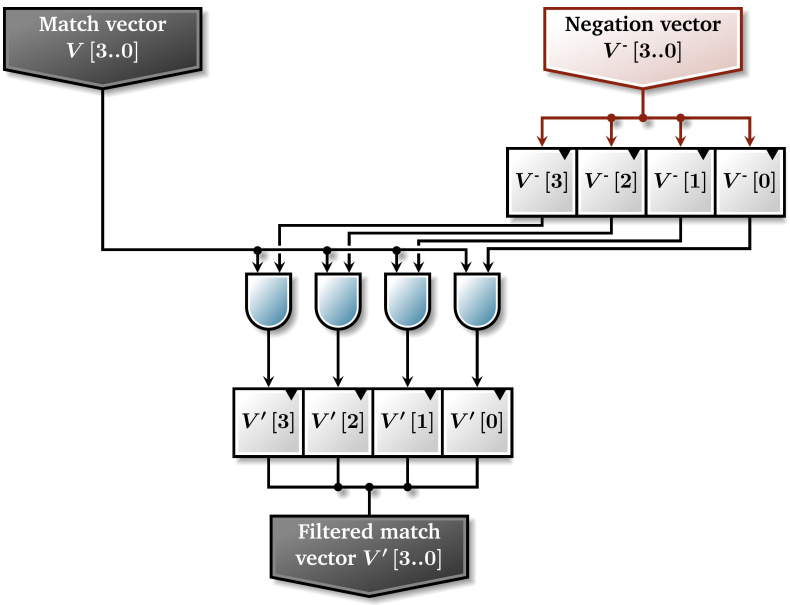


Fig. 16.3: Schematic of the negation vector circuit, which is used to disable rules within the utilized MPFC matcher.

MPFC circuitry by adding a pipelined negation vector component, as sketched in Figure 16.3. The negation vector is used to AND a runtime configurable mask $V^{\text{-}}$ onto the match vector V computed by the MPFC matcher to create a *reduced match vector* V'. As such, it is possible to disable any static rule R_i^{MPFC} by setting the corresponding bit $V'[i]$ in V' to zero. Subsequently, V', instead of V, is fed into the priority encoder in order to compute the index of the most highly prioritized matching static rule that is enabled. Therefore, any rule in $\mathfrak{R}^{\text{MPFC}}$ can quickly be dis- and enabled at runtime with a single write operation.

16.4 Performance Characteristics

In this section we summarize the performance and resource requirements of the proposed Consul approach in terms of processing latency, FFs, and combinational operations. As Consul is a hybrid classification approach which linearly arranges an optimized generic matcher M_{MPFC} and a generic matcher M_{gen} in a pipeline structure, Consul's processing latency equals the sum of the individual matchers' processing latencies, plus two additional cycles for the negation vector and action consolidation steps. It should be noted, though, that it is also possible to arrange the two matchers in parallel in order to reduce the latency. However, such a parallel matcher layout could result in a lower achievable clock frequency, as more combinational logic is packed in fewer pipeline stages.

The number of required configuration FFs is the sum of the number of bits $|\mathfrak{R}^{\text{MPFC}}|$ in the negation vector and the number of configuration FFs used by the generic matcher M_{gen}. Although the original MPFC approach does not require any configuration FFs at all, we argue that one bit per optimized rule can be justified by the potential large performance gain in the case of rule deletions. Furthermore, when considering the TCAM and StrideBV matchers, this sum is clearly dominated by the required amount of generic configuration bits even for moderate choices of $|\mathfrak{R}^{\text{gen}}|$.

The combinational operations that must be carried out during a classification operation in the Consul pipeline consist of the operations of both utilized matchers plus the operations executed in the negation vector and action consolidation modules. The number of actual matching operations can be represented as a linear combination between the products of the sub rule set sizes with the operations requirements of the corresponding matcher.

When it comes to support for range and negated checks, Consul inherits the native implementation support from MPFC for each rule implemented in $\mathfrak{R}^{\text{MPFC}}$. However, rules that are to be implemented by M_{gen} potentially need to be range-expanded, if the used generic matcher does not inherently support range or negated checks. Consul's support for range and negated checks, as well as the previously discussed performance characteristics, are summarized in Table 16.2, in comparison to related work.

16.5 Evaluation

We evaluate the Consul approach by inspecting the same hardware performance indicators we already used for our MPFC circuit evaluation in Section 15.5, namely the *circuit size* in terms of LUTs and FFs as well as the *circuit power consumption*. Moreover, we investigate the time required to create the search data structures, i. e., the contents of the configuration memories, of the utilized generic TCAM and StrideBV matchers used within the Consul pipeline.

16.5.1 Experiment Setup

The utilized hardware of the evaluation setup is identical to the setup described in Section 15.5, in which we target a Virtex 7 xc7vx690tffg1761-2 FPGA with our designs, with the packet processing pipeline clocked at 180 MHz. Our build machine, which runs a Fedora 20 Linux operating system, is equipped with an Intel Xeon E3-1270 CPU clocked at 3.50 GHz as well as 16 GB RAM. Again, we

Approach	Latency in cycles	Configuration flip-flops	Combinational operations	Supports range checks	Supports negated checks
Related work					
StrideBV [56]	$\lceil \frac{d \cdot w}{s} \rceil$	$2^s \cdot \lceil \frac{d \cdot w}{s} \rceil \cdot n$	$\left(\lceil \frac{d \cdot w}{s} \rceil - 1\right)$ n-bit ANDs	no	no
TCAM [66, 93]	1	$2 \cdot d \cdot w \cdot n$	$n \cdot (d \cdot w)$-bit eq. tests $2 \cdot n$ $(d \cdot w)$-bit ANDs	no	no
MPFC [Chp. 15]	1	0	$\mathcal{O}(d \cdot n)$ $\mathcal{O}(w)$-bit eq. tests $\mathcal{O}(n)$ $\mathcal{O}(d)$-bit ANDs	yes	yes
Proposed approach					
Consul	2 + lat_{MPFC} + lat_{gen}	$\text{ffs}_{\mathfrak{R}^{\text{gen}}}$ + $\lvert\mathfrak{R}^{\text{MPFC}}\rvert$	$\text{ops}_{\mathfrak{R}^{\text{MPFC}}}$, $\text{ops}_{\mathfrak{R}^{\text{gen}}}$, 1 comp., $\lvert\mathfrak{R}^{\text{MPFC}}\rvert$-bit AND, 1 mux	in $\mathfrak{R}^{\text{MPFC}}$	in $\mathfrak{R}^{\text{MPFC}}$

n: number of rules d: number of fields w: bits per field s: stride width

$\mathfrak{R}^T$: sub rule set implemented in approach T

lat_T: latency of approach T

$\text{ffs}_{\mathfrak{R}^T}$: number of configuration flip-flops in approach T when configured for rule set $\mathfrak{R}^T$

$\text{ops}_{\mathfrak{R}^T}$: combinational operations in approach T when configured for rule set $\mathfrak{R}^T$

Tab. 16.2: Consul performance characteristics, in comparison to related work (without priority encoder and action lookup).

use the self-written C++ tool `hardbit` in order to generate the entire Consul pipeline, which are the steps II and III in Figure 16.2. The packet preprocessing (Step I) and postprocessing (Step IV) are not included in the `hardbit` generation and the evaluation.

In order to evaluate the hardware resource footprint of the Consul pipeline, we use the benchmark generator *ClassBench* [132] to create ten different rule sets $\mathfrak{R}_{i,n}^{\text{MPFC}}$ for every rule set size $n \in \{1, 100, 200, \ldots, 1000\}$. These rule sets are used for the implementation of the specialized matcher M_{MPFC} (due to implementation details, M_{MPFC} must at least implement one rule). Also, for each rule set size n, we use ClassBench to generate ten rule sets $\mathfrak{R}_{i,1000-n}^{\text{gen}}$ of size $1000 - n$ to be used for the configuration of the generic matcher M_{gen}. Subsequently, for each n, we generate ten Consul pipelines $P_{i,n}$, whose specialized matcher M_{MPFC} implements $\mathfrak{R}_{i,n}^{\text{MPFC}}$ and whose generic matcher M_{gen} has a capacity for $1000 - n$ rules. Finally, we use Xilinx' *Vivado 2015.1* to synthesize, place, and route the $P_{i,n}$ designs and to extract the abovementioned hardware performance indicators from the implementation results.

To determine the time to create the search data structures for TCAM and StrideBV matchers that correspond to the rule sets $\mathfrak{R}_{i,1000-n}^{\text{gen}}$, we use a self-written C++

tool called `softbit`, which is the counterpart tool to `hardbit`. Both tools are compiled using the `g++ 4.8.3` compiler with the compile flags `-O0 -Wall -Werror -pedantic-errors -Wextra -std=c++0x -pthread`.

All data points shown in the following plots represent averaged results over ten evaluation runs, together with the corresponding 95% confidence intervals. Each evaluation run is executed on the above described build machine. We used the same 100 bit header as described in Section 15.5, and configured each StrideBV matcher width a stride width of ten bits.

16.5.2 Hardware Resource Footprint

Consul's hardware resource footprint in terms of LUTs and FFs is shown in Figure 16.4 and Figure 16.5, respectively. As expected, the figures demonstrate that the circuit size rises with an increasing capacity of the generic matcher. Since the Consul pipeline represents a linear combination of a specialized and a generic matcher, we observe a linear change in the circuit size when the capacities of M_{gen} and M_{MPFC} are modified. The figures also clearly illustrate the cost of runtime updateable matching circuitry: when switching from 1000 MPFC rules to 700 MPFC rules (thereby gaining a generic capacity of 300 rules), the LUT requirements already rise by factors of $7.8\times/2.1\times$ for TCAM/StrideBV configurations, respectively. In the case of FFs, these factors are $13.4\times/1.8\times$. When further increasing the size of the generic matcher to 700 rules, e. g., for use in a more dynamic environment, these factors increase to $16.9\times/3.3\times$ for LUTs and $29.8\times/2.4\times$ for FFs. Note that in the case of StrideBV, also further BRAM resources are required in order to store the bit vector search data structure.

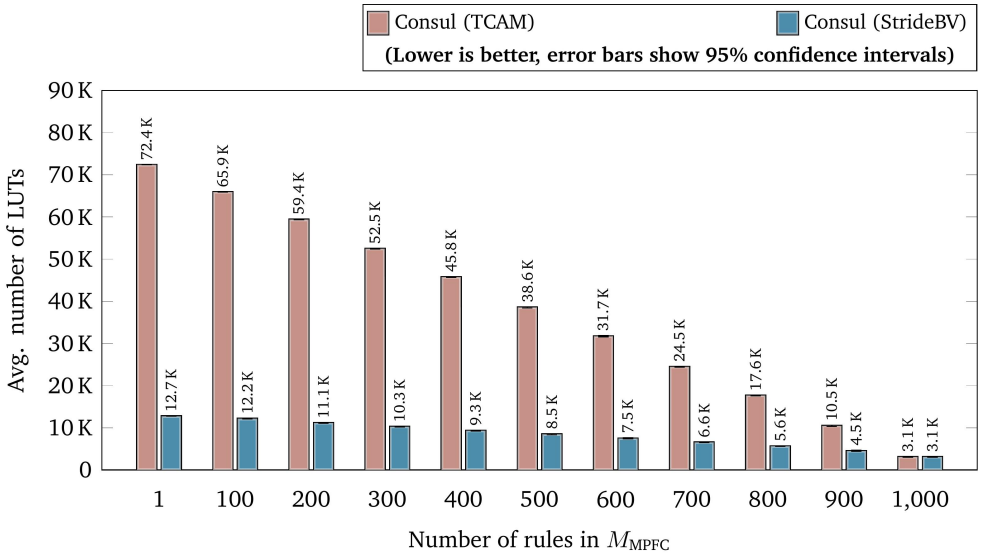


Fig. 16.4: Average LUT usage of the Consul pipeline with 1,000 rules.

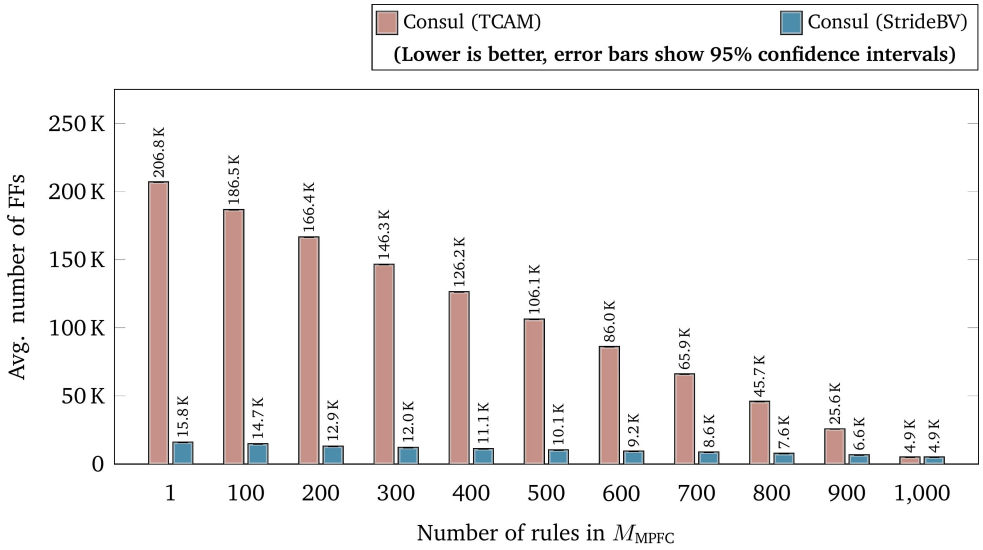


Fig. 16.5: Average FF usage of the Consul pipeline with 1,000 rules.

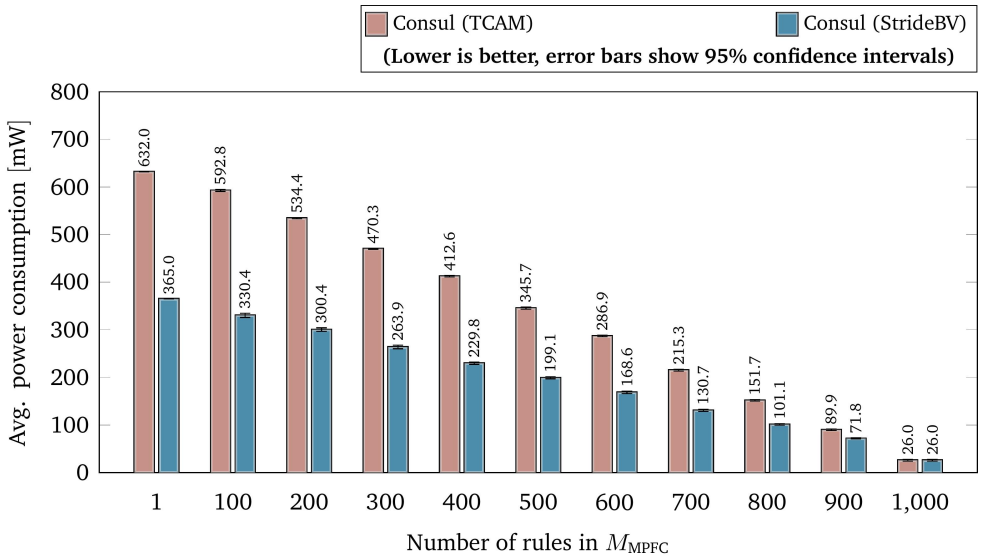


Fig. 16.6: Average power consumption of the Consul pipeline with 1,000 rules.

Figure 16.6 exhibits an analogous behaviour for the circuits' power consumption: as the circuit size grows, so does the corresponding power consumption.

16.5.3 Search Data Structure Computation

One of the main motivations for the Consul matcher is the ability to perform quick on-the-fly rule set updates at runtime, without the need for costly circuit synthesis runs. As such, in our second experiment, we investigate the amount of time required to compute the configuration memory contents of the utilized TCAM and StrideBV matchers for the generated test rule sets $\mathfrak{R}^{\text{gen}}_{i,1000-n}$. To this end, we

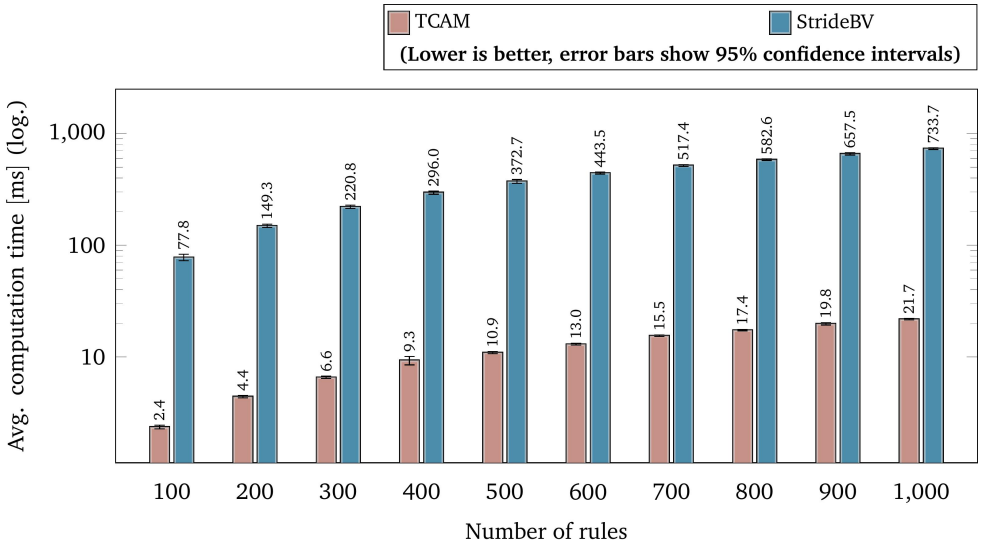


Fig. 16.7: Average `softbit` computation times for the search data structures utilized by the configuration memories of TCAM and StrideBV matchers.

measure the execution times of `softbit` to compute the complete search data structure for the TCAM and StrideBV matchers, which are shown in Figure 16.7.

It can be seen that, for both the TCAM and StrideBV matchers, the average search data structure creation time is well below one second for every regarded rule set size. Due to its simplicity, the TCAM contents are computed significantly faster than the StrideBV bit vectors, which makes the TCAM more amenable to dynamic and even incremental rule set updates. For its generic matcher M_{gen}, the Consul pipeline inherits these quick update capabilities and is thus able to quickly implement urgent rule set updates, either through insertions in M_{gen} or by disabling rules in M_{MPFC} by using the negation vector circuit. Of course, these updates also have to be written from the host system to the FPGA, which can be quickly executed by, e. g., Peripheral Component Interconnect Express (PCIe) burst transfers [182].

16.6 Limitations

The Consul approach is designed to alleviate the main issue of rule set specialized circuitry, namely the long rule set update durations. As discussed in Section 16.3, most update scenarios can be handled by using either the generic matcher M_{gen} or the negation vector module, which can both quickly be configured at system run time. Despite this fact, situations where the specialized matcher M_{MPFC} has to be rebuilt will still occur, and a classification system using the Consul pipeline must be able to handle them appropriately. One way to deal with such circuit rebuilds

would be to periodically schedule rebuilds on the host system in advance, when a certain load factor on the generic matcher is exceeded. This, in combination with a software- or FPGA-based classification fallback engine [9] that becomes active during the actual (partial) reconfiguration, would be a way to completely avoid situations at which the classification system is offline.

Another drawback a Consul system can suffer from is the fact that the generic matcher may not provide the same efficient rule representation capabilities as the specialized MPFC matcher, for example with respect to range or negated checks. Although it is possible to implement such rules in the generic matcher by using expansion techniques, this can lead to an inefficient rule representation within the configuration memories of the generic matcher [45].

17 Summary

Most existing hardware-based packet classification systems rely on generic matching circuitry which relies on configuration memories to process in coming network packets [56, 69, 109, 125]. In this chapter, we presented the fundamentally different approach MPFC, which exploits the inherent reconfigurability of FPGAs in order to embed a specified rule set into the circuit structure itself. To this end, rule sets are compiled into tailor-made matching circuits which only specify the exact semantics of one particular rule set. As a result, the specialized MPFC circuits are orders of magnitude more efficient in terms of hardware resource utilization and power dissipation than generic matching circuits of corresponding size, while maintaining the same throughput. In fact, the rule set specialized MPFC matchers are able to close the often observed performance gap between ASICs and FPGAs of one order of magnitude [37, 54], when assuming that ASICs are no suitable implementation platforms for rule set specialized matching circuits.

Our evaluation results for rule sets up to 1,024 rules demonstrate that MPFC circuits require up to 40× fewer LUTs than a TCAM and and up to 6× fewer LUTs than a StrideBV matcher. In terms of FF usage, these numbers increase to 211× and 10×, respectively. Moreover, our results exhibit that MPFC matchers inherently exploit check redundancy in the implemented rule set: the more often certain checks are used in the same dimension, the more efficient the circuits are optimized. Finally, we observed that additional rule set preprocessing can help to optimize the generated circuit structure even further by employing an exact rule set redundancy removal [84] *before* the circuit compilation. That way, we can aid the general-purpose heuristic logic minimization process used within the synthesis tool with domain-specific knowledge to generate equivalent, but smaller matchers.

Furthermore, we presented two techniques in order to make the MPFC approach scale for larger rule sets: tree-shaped and pipelined priority encoders and multi-stage MPFC circuits. Using these techniques, it is possible to operate MPFC circuits on a NetFPGA-SUME board for up to 5,000 rules at a clock frequency of 180 MHz, and a data bus width of 512 bits, which results in a maximum achievable throughput of 92.16 Gbit/s while solving the Geometric Packet Classification Problem [6] .

However, we also identified the major drawback of the MPFC approach, namely the long rule set implementation and update durations. There exist several approaches to address this issue: using partial reconfiguration and a double-buffered system, the matching circuit can be replaced at run time without the need to shut down the device [8, 9]. Another possibility is to utilize a hardware-software co-design, which can implement rule set updates in software, until circuit re-synthesis has finished [3, 6].

A further option is to leverage a hybrid matching circuit in order to combine the benefits of specialized and generic approaches, as presented in Chapter 16: the Consul approach. A Consul classification system arranges a highly optimized MPFC matcher and a generic matcher in a packet processing pipeline in order to combine the benefits of these approaches: while the MPFC matcher requires a low hardware resource footprint for static rules, the generic matcher provides the capability to implement dynamic rule set changes at system run time. Furthermore, Consul augments the specialized matcher with a run time programmable negation vector structure, which can be used to dynamically en- and disable hardwired rules at run time. Our evaluation results demonstrate that the additional adjustment screw provided by Consul, namely the partitioning of the rule set into static and dynamic sub rule sets, can be used to generate flexible classification systems for either static or dynamic environments, that still require significantly fewer hardware resources than a purely generic approach.

Epilog

18 Conclusion

In this work, we discussed the old but still highly relevant packet classification problem. We comprehensively reviewed existing approaches that belong to the three major directions to address packet classification, namely generic classification algorithms, rule set transformators, and hardware-assisted matchers. To each of these three classes, we contributed novel approaches that are able to improve upon certain key performance characteristics for classification systems, as detailed comparisons with a vast range of related work confirmed. All introduced ideas in this work are based on system specialization or hybrid techniques: while system specialization allows for high performance gains when taking certain traits of the underlying implementation platform into account, hybrid techniques combine the best features of two diametrically opposed systems. Table 18.1 summarizes our contributions alongside their specific traits and key features.

In the realm of classification algorithms, we contributed the *(Aggregated) Jit Vector Search* and *SFL* approaches. (Aggregated) Jit Vector Search builds upon the existing seminal (Aggregated) Bit Vector Search [31, 76], specializes the search data structure implementation on the utilized rule set, and exploits SIMD capabilities of the underlying CPU. As a result, (Aggregated) Jit Vector Search reaches near-optimal classification performance, while maintaining scalability with respect to rule set size. The SFL approach, on the other hand, is a hybrid classification technique, which is able to maintain high updateability while still utilizing the excellent classification performance of fast classification algorithms with static search data structures. As we have demonstrated, SFL is able to achieve superior classification and update rates in dynamic environments.

In order to improve the performance of practically used classification systems, such as firewalls [165, 167, 168], we devised the *RuleBender* transformation technique. The key idea behind RuleBender is to specialize rule sets on the jump semantics of the underlying classification system by encoding decision tree search data structures into the rule set itself, while maintaining its matching semantics both with respect to geometric checks and a vast range of complex checks. That way, existing systems that normally search the installed rule set linearly are able to benefit from the significantly faster traversal of decision trees. Furthermore, RuleBender can be turned into a hybrid transformator with even

Approach	Hybrid	Specialization-based	Key features
CLASSIFICATION ALGORITHMS			
(Aggregated) Jit Vector Search	no	yes	✓ close to optimal practical classification performance ✓ good scalability
SFL	yes	no	✓ high performance in dynamic environments ✓ support for stateless complex checks
RULE SET TRANSFORMATION			
RuleBender	yes	yes	✓ high performance increase for systems with jump semantics ✓ supports vast range of complex checks
HARDWARE-CENTRIC APPROACHES			
MPFC	no	yes	✓ line speed classification ✓ low hardware resource footprint ✓ exploits FPGA reconfigurability ✓ natively supports negated and range checks
Consul	yes	yes	✓ line speed classification ✓ combines advantages of two matchers with different traits

Tab. 18.1: Techniques introduced in this work alongside their specific traits, grouped by packet classification approach classes.

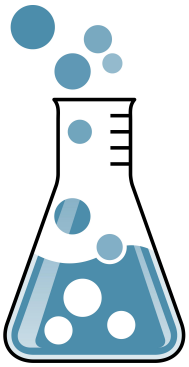
better performance through the combination with existing minimization-based transformators [71, 84]. We demonstrated that the application of RuleBender to the existing `iptables`, `nftables`, and `ipfw` systems can improve the achievable throughput by an order of magnitude.

With the *MPFC* and *Consul* matching pipelines, we presented techniques to solve the Geometric Packet Classification Problem on FPGA platforms in order to achieve line speed performance at a low hardware resource footprint. Due to MPFC's direct rule set to matching circuit translation, we measured an improvement of an order of magnitude in terms of utilized implementation resources in comparison with generic line speed matchers. As MPFC sacrifices updateability for highly optimized specialized matchers, we proposed the follow-up technique Consul, which combines the tailor-made MPFC matchers with an existing generic matcher. We have shown that the hybrid Consul matchers combine the advantages of both

approaches in form of still smaller resource requirements and the capability for quick updates at runtime.

Many approaches we proposed in this work are inspired by already existing techniques, which could be used to great effect when either combined or applied in a slightly different context. For example, the RuleBender transformator combines ideas that stem from decision tree classification algorithms as well as from existing transformators and combines them with the control flow mechanism in practical tools. The MPFC generator exploits the reconfiguration ability of FPGAs to generate partially evaluated matching circuits, that, again, can further be optimized by the utilization of rule set transformation techniques. Finally, the proposed hybrid Consul matching pipeline and SFL classification algorithm combine techniques with diametrically opposed performance characteristics, which in both cases results in responsive systems with high classification performance.

We conclude this work in the hope to have sown inspiration for future classification systems: *always remember great existing works, consider their strengths and weaknesses, take a look at the underlying platform, and think outside of the box*.



Abbreviations

A

ABV Aggregated Bit Vector 61

ACL Access Control List

ALM Adaptive Logic Module 150

ASIC Application-specific Integrated Circuit 146

AVX(2) Advanced Vector Extensions (2) 66

AVX-512 Advanced Vector Extensions 512 66

B

BDD Binary Decision Diagram 165

BPF Berkeley Packet Filter 60

BRAM Block RAM 151

BV Bit Vector 61

C

CAM Content-addressable Memory 155

CLB Configurable Logic Block 150

CPCP Complex Packet Classification Problem 19

CPU Central Processing Unit 27

CRR Complete Redudancy Removal 112

Consul Hybrid Classification Circuit with Action Consolidation 193

D

DNS Domain Name System 10

DOS Denial of Service 194

DPI Deep Packet Inspection 165

E

F

G

H

I

J

L

M

O

P

Q

R

S

T

U

V

Bibliography

(Co-)Authored Peer-reviewed Publications

[1] S. Brack, S. Hager, and B. Scheuermann. „JitVector: Just-in-Time Code Generation for Network Packet Classification“. In: *LCN ’15: Proceedings of the 40th Annual IEEE International Conference on Local Computer Networks*. Clearwater Beach, FL, USA, Oct. 2015, pp. 161–164 (cit. on pp. 5, 27).

[2] A. Fießler, S. Hager, and B. Scheuermann. „Flexible Line Speed Network Packet Classification Using Hybrid On-chip Matching Circuits“. In: *HPSR ’17: Proceedings of the 2017 IEEE 18th International Conference on High Performance Switching and Routing*. Campinas, Brazil, June 2017, pp. 1–8 (cit. on pp. 7, 79, 145, 186).

[3] A. Fießler, S. Hager, B. Scheuermann, and A. Moore. „HyPaFilter - A Versatile Hybrid FPGA Packet Filter“. In: *ANCS ’16: Proceedings of the 2016 ACM/IEEE Ninth Symposium on Architectures for Networking and Communication Systems*. Santa Clara, CA, USA, Mar. 2016, pp. 25–36 (cit. on pp. 9, 79, 141, 145, 147, 183, 210).

[4] A. Fießler, D. Loebenberger, S. Hager, and B. Scheuermann. „On the Use of (Non-)Cryptographic Hashes on FPGAs“. In: *ARC ’17: Proceedings of the 13th International Symposium on Applied Reconfigurable Computing*. Delft, The Netherlands, Apr. 2017, pp. 72–80 (cit. on p. 146).

[5] A. Fießler, C. Lorenz, S. Hager, and B. Scheuermann. „FireFlow – High Performance Hybrid SDN-Firewalls with OpenFlow“. In: *LCN ’18: Proceedings of the 43rd Annual IEEE International Conference on Local Computer Networks*. Chicago, IL, USA, Oct. 2018, pp. 227–230 (cit. on p. 141).

[6] A. Fießler, C. Lorenz, S. Hager, B. Scheuermann, and A. Moore. „HyPaFilter+: Enhanced Hybrid Packet Filtering Using Hardware Assisted Classification and Header Space Analysis“. In: *IEEE/ACM Transactions on Networking* 25.6 (Dec. 2017), pp. 3655–3669. ISSN: 1063-6692 (cit. on pp. 9, 141, 145, 147, 152, 183, 209, 210).

[7] W. Gusew, S. Hager, and B. Scheuermann. „CATE: An Open and Highly Configurable Framework for Performance Evaluation of Packet Classification Algorithms“. In: *WSC ’15: Proceedings of the 2015 Winter Simulation Conference*. Huntington Beach, CA, USA, Dec. 2015, pp. 3037–3048 (cit. on p. 27).

[8] S. Hager, D. Bendyk, and B. Scheuermann. „Matching circuits can be small: Partial evaluation and reconfiguration for FPGA-based packet processing“. In: *Journal of Parallel and Distributed Computing* 109.C (Nov. 2017), pp. 42–49. ISSN: 0743-7315 (cit. on pp. 7, 145, 192, 201, 210).

[9] S. Hager, D. Bendyk, and B. Scheuermann. „Partial Reconfiguration And Specialized Circuitry for Flexible FPGA-based Packet Processing“. In: *ReConFig '15: Proceedings of the 2015 International Conference on ReConFigurable Computing and FPGAs*. Cancun, Mexico, Dec. 2015, pp. 1–6 (cit. on pp. 7, 145, 192, 208, 210).

[10] S. Hager, S. Brack, and B. Scheuermann. „The Small, the Fast, and the Lazy (SFL): A General Approach for Fast and Flexible Packet Classification“. In: *LCN '16: Proceedings of the 41th Annual IEEE International Conference on Local Computer Networks*. Dubai, UAE, Nov. 2016, pp. 43–51 (cit. on pp. 5, 27, 38, 43, 47, 61, 107).

[11] S. Hager, P. John, S. Dietzel, and B. Scheuermann. „RuleBender: Tree-based policy transformations for practical packet classification systems“. In: *Computer Networks* 135.1 (Apr. 2018), pp. 253–265. ISSN: 1389-1286 (cit. on pp. 6, 9, 107).

[12] S. Hager, P. John, A. Fießler, and B. Scheuermann. „Minflate: Combining Rule Set Minimization with Jump-based Expansion for Fast Packet Classification“. In: *ANCS '16: Proceedings of the 2016 ACM/IEEE Ninth Symposium on Architectures for Networking and Communication Systems*. Santa Clara, CA, USA, Mar. 2016, pp. 115–116 (cit. on pp. 6, 79, 107).

[13] S. Hager, S. Selent, and B. Scheuermann. „Trees in the List: Accelerating List-based Packet Classification Through Controlled Rule Set Expansion“. In: *CoNEXT '14: Proceedings of the 10th International Conference on Emerging Networking Experiments and Technologies*. Sydney, Australia, Dec. 2014, pp. 101–107 (cit. on pp. 6, 9, 33, 79, 107, 128).

[14] S. Hager, F. Winkler, B. Scheuermann, and K. Reinhardt. „Building Optimized Packet Filters with COFFi“. In: *FCCM '14: Proceedings of the 22nd IEEE International Symposium on Field-Programmable Custom Computing Machines*. Boston, MA, USA, May 2014, pp. 105–105 (cit. on pp. 7, 145).

[15] S. Hager, F. Winkler, B. Scheuermann, and K. Reinhardt. „MPFC: Massively Parallel Firewall Circuits“. In: *LCN '14: Proceedings of the 39th Annual IEEE International Conference on Local Computer Networks*. Edmonton, Canada, Sept. 2014, pp. 305–313 (cit. on pp. 7, 39, 79, 145, 146, 178).

(Co-)Authored Domestic Publications

[16] A. Fießler, S. Hager, B. Scheuermann, and A. von Gernler. „HardFIRE - ein Firewall-Konzept auf FPGA-Basis“. In: *14. Deutscher BSI-Sicherheitskongress*. Bonn, Germany, May 2015, pp. 251–262 (cit. on p. 145).

[17] S. Hager, A. Fießler, and B. Scheuermann. *Fast Firewalling Through System Specialization*. Abstract and presentation at ITG-NHNS Workshop, Sep. 2015: `https://kn.inf.uni-tuebingen.de/ITG-NHNS-2015`. Last access: Feb. 1, 2019 (cit. on pp. 5, 145).

[18] C. Lorenz, A. Fießler, and S. Hager. „FlowFire: Beschleunigung komplexer Firewall-Appliances mittels Software Defined Networking“. In: *16. Deutscher BSI-Sicherheitskongress*. Bonn, Germany, May 2019, pp. 157–166 (cit. on p. 141).

Theses Supervised by the Author

[19] D. Bendyk. „On the Applicability of Partial Reconfiguration for the Efficient Representation of Routing Tables in FPGAs“. Bachelor's thesis. Humboldt-Universität zu Berlin, 2015 (cit. on p. 145).

[20] S. Brack. „Hybrid Packet Classification“. Master's thesis. Humboldt-Universität zu Berlin, 2016 (cit. on p. 27).

[21] S. Brack. „Measurements and Optimizations with Just-In-Time Code Generation on the OpenFlow Reference Implementation“. Bachelor's thesis. Humboldt-Universität zu Berlin, 2014 (cit. on p. 27).

[22] W. Gusew. „Benchmarking Classification Algorithms for Packet Filtering with CATE“. Master's thesis. Humboldt-Universität zu Berlin, 2014 (cit. on p. 27).

[23] P. John. „Evaluation of Rule Set Optimizations for Achieving High-Performance Software-based Packet Classification“. Diploma thesis. Humboldt-Universität zu Berlin, 2016 (cit. on p. 107).

[24] S. Selent. „Firewall Ruleset Transformations for iptables“. Bachelor's thesis. Humboldt-Universität zu Berlin, 2014 (cit. on p. 107).

Peer-reviewed Publications

[25] K. Accardi, T. Bock, F. Hady, and J. Krueger. „Network Processor Acceleration for a Linux* Netfilter Firewall“. In: *ANCS '05: Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems*. Princeton, NJ, USA, Oct. 2005, pp. 115–123 (cit. on p. 141).

[26] S. Acharya, J. Wang, Z. Ge, T. Znati, and A. Greenberg. „Traffic-Aware Firewall Optimization Strategies“. In: *ICC '06: Proceedings of the 2006 IEEE International Conference on Communications*. Istanbul, Turkey, June 2006, pp. 2225–2230 (cit. on p. 117).

[27] A. Ahmed, K. Park, and S. Baeg. „Resource-Efficient SRAM-Based Ternary Content Addressable Memory“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.4 (Apr. 2017), pp. 1583–1587. ISSN: 1063-8210 (cit. on p. 155).

[28] B. Alpern, M. Wegman, and F. Zadeck. „Detecting Equality of Variables in Programs“. In: *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, CA, USA, Jan. 1988, pp. 1–11 (cit. on p. 60).

[29] M. Attig and G. Brebner. „400 Gb/s Programmable Packet Parsing on a Single FPGA“. In: *ANCS '11: Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*. Brooklyn, NY, USA, Oct. 2011, pp. 12–23 (cit. on pp. 145, 166).

[30] F. Baboescu, S. Singh, and G. Varghese. „Packet Classification for Core Routers: Is there an alternative to CAMs? “ In: *INFOCOM '03: Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*. San Francisco, CA, USA, Mar. 2003, pp. 53–63 (cit. on pp. 3, 155).

[31] F. Baboescu and G. Varghese. „Scalable Packet Classification“. In: *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. San Diego, CA, USA, Aug. 2001, pp. 199–210 (cit. on pp. 3, 4, 14, 27–29, 31, 36, 39–41, 48, 70, 79, 103, 213).

[32] V. Bala, E. Duesterwald, and S. Banerjia. „Dynamo: A Transparent Dynamic Optimization System“. In: *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. Vancouver, British Columbia, Canada, June 2000, pp. 1–12 (cit. on p. 117).

[33] A. Begel, S. McCanne, and S. Graham. „BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture“. In: *SIGCOMM '99: Proceedings of the 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Cambridge, MA, USA, Aug. 1999, pp. 123–134 (cit. on p. 60).

[34] M. Bertrone, S. Miano, F. Risso, and M. Tumolo. „Accelerating Linux Security with eBPF Iptables“. In: *SIGCOMM '18: Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. Budapest, Hungary, Aug. 2018, pp. 108–110 (cit. on pp. 60, 61, 77).

[35] S. Bhandari, S. Subbaraman, S. Pujari, and R. Mahajan. „Real Time Video Processing on FPGA Using on the Fly Partial Reconfiguration“. In: *ICSPS '09: Proceedings of the 2009 International Conference on Signal Processing Systems*. Singapore, Singapore, May 2009, pp. 244–247 (cit. on p. 146).

[36] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. „P4: Programming Protocol-Independent Packet Processors“. In: *SIGCOMM Computer Communication Review* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833 (cit. on p. 166).

[37] P. Bosshart, G. Gibb, H. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. „Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN“. In: *SIGCOMM '13: Proceedings of the 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Hong Kong, China, Aug. 2013, pp. 99–110 (cit. on pp. 146, 155, 169, 209).

[38] G. Brebner and W. Jiang. „High-Speed Packet Processing using Reconfigurable Computing“. In: *IEEE Micro* 34.1 (Jan. 2014), pp. 8–18. ISSN: 0272-1732 (cit. on p. 167).

[39] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Hel-Or. „Encoding Short Ranges in TCAM Without Expansion: Efficient Algorithm and Applications“. In: *SPAA '16: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. Pacific Grove, CA, USA, July 2016, pp. 35–46 (cit. on pp. 158, 176).

[40] A. Bremler-Barr and D. Hendler. „Space-Efficient TCAM-Based Classification Using Gray Coding“. In: *IEEE Transactions on Computers* 61.1 (Jan. 2012), pp. 18–30. ISSN: 0018-9340 (cit. on pp. 158, 176).

[41] R. Bryant. „Graph-Based Algorithms for Boolean Function Manipulation“. In: *IEEE Transactions on Computers* C-35.8 (Aug. 1986), pp. 677–691. ISSN: 0018-9340 (cit. on p. 165).

[42] S. Chandrakar, D. Gaitonde, and T. Bauer. „Enhancements in UltraScale CLB Architecture“. In: *FPGA '15: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey, California, USA, Feb. 2015, pp. 108–116 (cit. on p. 150).

[43] D. Chang and P. Wang. „TCAM-Based Multi-Match Packet Classification Using Multidimensional Rule Layering“. In: *IEEE/ACM Transactions on Networking* 24.2 (Apr. 2016), pp. 1125–1138. ISSN: 1063-6692 (cit. on pp. 128, 158).

[44] P. Chang and W. Hwu. „Inline Function Expansion for Compiling C Programs“. In: *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. Portland, Oregon, USA, June 1989, pp. 246–257 (cit. on p. 125).

[45] Y. Chang. „A 2-Level TCAM Architecture for Ranges“. In: *IEEE Transactions on Computers* 55.12 (Dec. 2006), pp. 1614–1629. ISSN: 0018-9340 (cit. on pp. 107, 174, 176, 189, 208).

[46] H. Chen, Y. Chen, and D. Summerville. „A Survey on the Application of FPGAs for Network Infrastructure Security“. In: *IEEE Communications Surveys & Tutorials* 13.4 (Aug. 2011), pp. 541–561. ISSN: 1553-877X (cit. on pp. 145, 146, 165).

[47] J. Cocke. „Global Common Subexpression Elimination“. In: *ACM SIGPLAN Notices - Proceedings of a symposium on Compiler optimization* 5.7 (July 1970), pp. 20–24. ISSN: 0362-1340 (cit. on pp. 175, 187).

[48] J. Cong and K. Minkovich. „Optimality Study of Logic Synthesis for LUT-Based FPGAs“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (Feb. 2007), pp. 230–239. ISSN: 0278-0070 (cit. on pp. 152, 153, 175).

[49] J. Daly, A. Liu, and E. Torng. „A Difference Resolution Approach to Compressing Access Control Lists“. In: *IEEE/ACM Transactions on Networking* 24.1 (Feb. 2016), pp. 610–623. ISSN: 1063-6692 (cit. on pp. 5, 23, 79, 108, 115, 158).

[50] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. „Deep Packet Inspection using Parallel Bloom Filters“. In: *IEEE Micro* 24.1 (Jan. 2004), pp. 52–61. ISSN: 0272-1732 (cit. on p. 166).

[51] W. Doeringer, G. Karjoth, and M. Nassehi. „Routing on Longest-Matching Prefixes“. In: *IEEE/ACM Transactions on Networking* 4.1 (Feb. 1996), pp. 86–97. ISSN: 1063-6692 (cit. on p. 48).

[52] R. Draves, C. King, S. Venkatachary, and B. Zill. „Constructing Optimal IP Routing Tables“. In: *INFOCOM '99: Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*. New York, NY, USA, Mar. 1999, 88–97 vol.1 (cit. on pp. 5, 158).

[53] D. Engler and M. Kaashoek. „DPF: Fast, Flexible Message Demultiplexing Using Dynamic Code Generation“. In: *SIGCOMM '96: Proceedings of the 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Palo Alto, CA, USA, Aug. 1996, pp. 53–59 (cit. on p. 60).

[54] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. Maltz, and A. Greenberg. „Azure Accelerated Networking: SmartNICs in the Public Cloud“. In: *NSDI '18: Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, USA, Apr. 2018, pp. 51–66 (cit. on pp. 146, 209).

[55] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang. „ParaSplit: A Scalable Architecture on FPGA for Terabit Packet Classification“. In: *HOTI '12: Proceedings of the 2012 IEEE 20th Annual Symposium on High-Performance Interconnects*. Santa Clara, CA, USA, Aug. 2012, pp. 1–8 (cit. on p. 50).

[56] T. Ganegedara and V. Prasanna. „StrideBV: Single Chip 400G+ Packet Classification“. In: *HPSR '12: Proceedings of the 13th International Conference on High Performance Switching and Routing*. Belgrade, Serbia, June 2012, pp. 1–6 (cit. on pp. 6, 39, 79, 146, 149, 152, 155, 159, 162, 176, 177, 181, 193, 200, 201, 204, 209).

[57] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. „Design Principles for Packet Parsers“. In: *ANCS '13: Proceedings of the 2013 ACM/IEEE Ninth Symposium on Architectures for Networking and Communications Systems*. San Jose, CA, USA, Oct. 2013, pp. 13–24 (cit. on p. 166).

[58] X. Gong, W. Wang, and S. Cheng. „ERFC: An Enhanced Recursive Flow Classification Algorithm“. In: *Journal of Computer Science and Technology* 25.5 (Sept. 2010), pp. 958–969. ISSN: 1860-4749 (cit. on pp. 4, 27, 28, 48, 56, 71, 127).

[59] M. Gouda and A. Liu. „Structured firewall design“. In: *Computer Networks* 51.4 (Mar. 2007), pp. 1106–1120. ISSN: 1389-1286 (cit. on pp. 10, 112, 114).

[60] P. Gupta, S. Lin, and N. McKeown. „Routing Lookups in Hardware at Memory Access Speeds“. In: *INFOCOM '98: Proceedings of the 17th Annual Joint Conference of the IEEE Computer and Communications Societies*. San Francisco, CA, USA, Mar. 1998, 1240–1247 vol.3 (cit. on p. 48).

[61] P. Gupta and N. McKeown. „Algorithms for Packet Classification“. In: *IEEE Network: The Magazine of Global Internetworking* 15.2 (Mar. 2001), pp. 24–32. ISSN: 0890-8044 (cit. on pp. 3, 9, 29, 31, 47, 56, 79, 127).

[62] P. Gupta and N. McKeown. „Packet Classification on Multiple Fields“. In: *SIGCOMM '99: Proceedings of the 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Cambridge, Massachusetts, USA, Aug. 1999, pp. 147–160 (cit. on pp. 3–5, 21, 27–29, 42, 43, 61, 62, 70, 77, 79, 95, 96, 103).

[63] P. Gupta and N. McKeown. „Packet Classification using Hierarchical Intelligent Cuttings“. In: *HOTI '99: Proceedings of the 7th Symposium on High Performance Interconnects*. Stanford, CA, USA, Aug. 1999, pp. 34–41 (cit. on pp. 3, 4, 6, 27, 29, 31, 33, 50, 51, 53–55, 70, 77, 79, 103, 108, 120, 141).

[64] I. Hadžić and J. Smith. „P4: A Platform for FPGA Implementation of Protocol Boosters“. In: *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. London, United Kingdom, Sept. 1997, pp. 438–447 (cit. on p. 167).

[65] H. Hamed and E. Al-Shaer. „Dynamic Rule-ordering Optimization for High-speed Firewall Filtering“. In: *ASIACCS '06: Proceedings of the 2006 ACM Symposium on InformAtion, Computer and Communications Security*. Taipei, Taiwan, Mar. 2006, pp. 332–342 (cit. on pp. 79, 117).

[66] A. Hanlon. „Content-Addressable and Associative Memory Systems a Survey“. In: *IEEE Transactions on Electronic Computers* EC-15.4 (Aug. 1966), pp. 509–521. ISSN: 0367-7508 (cit. on pp. 6, 157, 181, 193, 204).

[67] W. Jiang. „Scalable Ternary Content Addressable Memory Implementation Using FPGAs“. In: *ANCS '13: Proceedings of the 2013 ACM/IEEE Ninth Symposium on Architectures for Networking and Communications Systems*. San Jose, CA, USA, Oct. 2013, pp. 71–82 (cit. on p. 155).

[68] W. Jiang and V. Prasanna. „Field-Split Parallel Architecture for High Performance Multi-match Packet Classification Using FPGAs“. In: *SPAA '09: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. Calgary, AB, Canada, Aug. 2009, pp. 188–196 (cit. on pp. 6, 39, 146, 149, 155, 165, 201).

[69] W. Jiang and V. Prasanna. „Scalable Packet Classification on FPGA“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20.9 (Sept. 2012), pp. 1668–1680. ISSN: 1063-8210 (cit. on pp. 6, 128, 145, 146, 155, 176, 209).

[70] L. Jose, L. Yan, G. Varghese, and N. McKeown. „Compiling Packet Programs to Reconfigurable Switches“. In: *NSDI '15: Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*. Oakland, CA, USA, May 2015, pp. 103–115 (cit. on p. 166).

[71] T. Katic and P. Pale. „Optimization of Firewall Rules“. In: *ITI '07: Proceedings of the 29th International Conference on Information Technology Interfaces*. Cavtat, Croatia, June 2007, pp. 685–690 (cit. on pp. 5, 21, 23, 108, 109, 111, 123, 126–128, 141, 158, 214).

[72] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster. „SAX-PAC (Scalable And eXpressive PAcket Classification)“. In: *SIGCOMM '14: Proceedings of the 2014 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Chicago, Illinois, USA, Aug. 2014, pp. 15–26 (cit. on p. 14).

[73] D. Kreutz, F. Ramos, P. Veríssimo, C. Rothenberg, S. Azodolmolky, and S. Uhlig. „Software-Defined Networking: A Comprehensive Survey“. In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76. ISSN: 0018-9219 (cit. on pp. 3, 186, 192).

[74] C. Kruegel and T. Toth. „Using Decision Trees to Improve Signature-Based Intrusion Detection“. In: *RAID '03: Proceedings of the 2003 International Symposium on Research in Attacks, Intrusions and Defenses*. Pittburgh, PA, USA, Sept. 2003, pp. 173–191 (cit. on pp. 3, 17).

[75] C. Labovitz, G. Malan, and F. Jahanian. „Internet Routing Instability“. In: *IEEE/ACM Transactions on Networking* 6.5 (Oct. 1998), pp. 515–528. ISSN: 1063-6692 (cit. on p. 192).

[76] T. Lakshman and D. Stiliadis. „High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching“. In: *SIGCOMM '98: Proceedings of the 1998 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Vancouver, BC, Canada, Aug. 1998, pp. 203–214 (cit. on pp. 3–5, 27–29, 31, 33, 36, 62, 63, 65, 69, 70, 79, 95, 96, 103, 213).

[77] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. „Algorithms for Advanced Packet Classification with Ternary CAMs“. In: *SIGCOMM '05: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Philadelphia, PA, USA, Aug. 2005, pp. 193–204 (cit. on pp. 158, 176).

[78] H. Lam, D. Wang, and H. Chao. „A Traffic-aware Top-N Firewall Approximation Algorithm“. In: *SCNC '11: Proceedings of the 1st International Workshop on Security in Computers, Networking and Communications*. Shanghai, China, Apr. 2011, pp. 1036–1041 (cit. on p. 117).

[79] T. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu, and N. Dulay. „Compiling Policy Descriptions into Reconfigurable Firewall Processors“. In: *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa, CA, USA, Apr. 2003, pp. 39–48 (cit. on p. 159).

[80] T. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu, and N. Dulay. „Irregular Reconfigurable CAM Structures for Firewall Applications“. In: *FPL '03: Proceedings of the 13th International Conference on Field Programmable Logic and Applications*. Lisbon, Portugal, Sept. 2003, pp. 890–899 (cit. on pp. 149, 159).

[81] W. Li, D. Li, Y. Bai, W. Le, and H. Li. „Memory-efficient recursive scheme for multi-field packet classification“. In: *IET Communications* 13.9 (Apr. 2019), pp. 1319–1325. ISSN: 1751-8628 (cit. on pp. 27, 28, 71).

[82] W. Li, X. Li, H. Li, and G. Xie. „CutSplit: A Decision-Tree Combining Cutting and Splitting for Scalable Packet Classification“. In: *INFOCOM '18: Proceedings of the 37th Annual Joint Conference of the IEEE Computer and Communications Societies*. Honolulu, HI, USA, Apr. 2018, pp. 2645–2653 (cit. on p. 29).

[83] E. Liang, H. Zhu, X. Jin, and I. Stoica. „Neural Packet Classification“. In: *SIGCOMM '19: Proceedings of the ACM Special Interest Group on Data Communication*. Beijing, China, Aug. 2019, pp. 256–269 (cit. on pp. 3, 27, 29, 50, 71).

[84] A. Liu and M. Gouda. „Complete Redundancy Detection in Firewalls“. In: *Proceedings of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*. Storrs, CT, USA, Aug. 2005, pp. 193–206 (cit. on pp. 5, 9, 23, 79, 107–109, 112, 123, 126–128, 141, 158, 190, 209, 214).

[85] A. Liu and M. Gouda. „Diverse Firewall Design“. In: *IEEE Transactions on Parallel and Distributed Systems* 19.9 (Sept. 2008), pp. 1237–1251. ISSN: 1045-9219 (cit. on pp. 21, 112, 114).

[86] A. Liu, C. Meiners, and E. Torng. „TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs“. In: *IEEE/ACM Transactions on Networking* 18.2 (Apr. 2010), pp. 490–500. ISSN: 1063-6692 (cit. on pp. 5, 115, 158).

[87] A. Liu, E. Torng, and C. Meiners. „Compressing Network Access Control Lists“. In: *IEEE Transactions on Parallel and Distributed Systems* 22.12 (Dec. 2011), pp. 1969–1977. ISSN: 1045-9219 (cit. on p. 115).

[88] A. Liu, E. Torng, and C. Meiners. „Firewall Compressor: An Algorithm for Minimizing Firewall Policies“. In: *INFOCOM '08: Proceedings of the 27th Annual Joint Conference of the IEEE Computer and Communications Societies*. Phoenix, AZ, USA, Apr. 2008, pp. 691–699 (cit. on pp. 5, 9, 21, 23, 79, 108, 109, 115, 116, 123, 126–128, 141, 158).

[89] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch. „Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration“. In: *FPL '09: Proceedings of the 2009 International Conference on Field Programmable Logic and Applications*. Prague, Czech Republic, Aug. 2009, pp. 498–502 (cit. on p. 192).

[90] Z. Liu, X. Wang, B. Yang, and J. Li. „BitCuts: Towards Fast Packet Classification for Order-Independent Rules“. In: *SIGCOMM '15: Proceedings of the 2015 Conference on Special Interest Group on Data Communication*. London, United Kingdom, Aug. 2015, pp. 339–340 (cit. on p. 50).

[91] R. Manimegalai, E. Siva Soumya, V. Muralidharan, B. Ravindran, V. Kamakoti, and D. Bhatia. „Placement and Routing for 3D-FPGAs using Reinforcement Learning and Support Vector Machines“. In: *VLSID '05: Proceedings of the 18th International Conference on VLSI Design*. Kolkata, India, Jan. 2005, pp. 451–456 (cit. on p. 153).

[92] J. Matoušek, G. Antichi, A. Lučanský, A. Moore, and J. Kořenek. „ClassBench-ng: Recasting ClassBench After a Decade of Network Evolution“. In: *ANCS '17: Proceedings of the Symposium on Architectures for Networking and Communications Systems*. Beijing, China, May 2017, pp. 204–216 (cit. on p. 128).

[93] A. McAuley and P. Francis. „Fast Routing Table Lookup Using CAMs“. In: *INFOCOM '93: Proceedings of the 12th Annual Joint Conference of the IEEE Computer and Communications Societies*. San Francisco, CA, USA, Mar. 1993, 1382–1391 vol.3 (cit. on pp. 156, 157, 181, 193, 200, 201, 204).

[94] S. McCanne and V. Jacobson. „The BSD Packet Filter: A New Architecture for User-level Packet Capture“. In: *USENIX '93: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. San Diego, CA, USA, Jan. 1993, pp. 259–269 (cit. on pp. 4, 60).

[95] E. McDonald. „Runtime FPGA Partial Reconfiguration“. In: *Proceedings of the 2008 IEEE Aerospace Conference*. Big Sky, MT, USA, Mar. 2008, pp. 1–7 (cit. on p. 146).

[96] A. McEwan and J. Saul. „A High Speed Reconfigurable Firewall Based On Parameterizable FPGA-based Content Addressable Memories“. In: *The Journal of Supercomputing* 19.1 (May 2001), pp. 93–103. ISSN: 1573-0484 (cit. on p. 159).

[97] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. „OpenFlow: Enabling Innovation in Campus Networks“. In: *SIGCOMM Computer Communication Review* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833 (cit. on pp. 9, 139).

[98] L. McMurchie and C. Ebeling. „PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs“. In: *FPGA '95: Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays*. Monterey, CA, USA, Feb. 1995, pp. 111–117 (cit. on pp. 151, 179).

[99] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. Bernal. „Creating Complex Network Services with eBPF: Experience and Lessons Learned“. In: *HPSR '18: Proceedings of the 2018 IEEE 19th International Conference on High Performance Switching and Routing*. Bucharest, Romania, June 2018, pp. 1–8 (cit. on pp. 4, 59, 60, 77).

[100] J. Mirkovic and P. Reiher. „A Taxonomy of DDoS Attack and DDoS Defense Mechanisms“. In: *SIGCOMM Computer Communication Review* 34.2 (Apr. 2004), pp. 39–53. ISSN: 0146-4833 (cit. on p. 194).

[101] J. Mogul, R. Rashid, and M. Accetta. „The Packet Filter: An Efficient Mechanism for User-level Network Code“. In: *SOSP '87: Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. Austin, TX, USA, Nov. 1987, pp. 39–51 (cit. on pp. 13, 59, 60).

[102] K. Pagiamtzis and A. Sheikholeslami. „Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey“. In: *IEEE Journal of Solid-State Circuits* 41.3 (Mar. 2006), pp. 712–727. ISSN: 0018-9200 (cit. on pp. 155, 156).

[103] K. Papagiannaki, N. Taft, S. Bhattacharyya, P. Thiran, K. Salamatian, and C. Diot. „A Pragmatic Definition of Elephants in Internet Backbone Traffic“. In: *IWM '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*. Marseille, France, Nov. 2002, pp. 175–176 (cit. on p. 201).

[104] V. Paxson. „Bro: A System for Detecting Network Intruders in Real-time“. In: *Computer Networks: The International Journal of Computer and Telecommunications Networking* 31.23-24 (Dec. 1999), pp. 2435–2463. ISSN: 1389-1286 (cit. on p. 9).

[105] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. „The Design and Implementation of Open vSwitch“. In: *NSDI '15: Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*. Oakland, CA, USA, May 2015, pp. 117–130 (cit. on pp. 4, 27–29, 34, 71, 79, 80).

[106] V. Puš and J. Kořenek. „Fast and Scalable Packet Classification Using Perfect Hash Functions“. In: *FPGA '09: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, California, USA, Feb. 2009, pp. 229–236 (cit. on p. 166).

[107] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. „Packet Classification Algorithms: From Theory to Practice“. In: *INFOCOM '09: Proceedings of the 28nd Annual Joint Conference of the IEEE Computer and Communications Societies*. Rio de Janeiro, Brazil, Apr. 2009, pp. 648–656 (cit. on pp. 4, 6, 27, 29, 31, 50, 51, 56, 59, 70, 77, 79, 95, 96, 108, 120, 127, 128, 141).

[108] Z. Qian and M. Margala. „Low Power RAM-based Hierarchical CAM on FPGA“. In: *ReConFig '14: Proceedings of the 2014 International Conference on ReConFigurable Computing and FPGAs*. Cancun, Mexico, Dec. 2014, pp. 1–4 (cit. on p. 155).

[109] W. Qu and V. Prasanna. „High-Performance and Dynamically Updatable Packet Classification Engine on FPGA“. In: *IEEE Transactions on Parallel and Distributed Systems* 27.1 (Jan. 2015), pp. 197–209. ISSN: 1045-9219 (cit. on pp. 6, 14, 146, 149, 152, 155, 162, 165, 176, 201, 209).

[110] Y. Qu, H. Zhang, S. Zhou, and V. Prasanna. „Optimizing Many-field Packet Classification on FPGA, Multi-core General Purpose Processor, and GPU“. In: *ANCS '15: Proceedings of the 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. Oakland, CA, USA, May 2015, pp. 87–98 (cit. on p. 145).

[111] H. Rice. „Classes of Recursively Enumerable Sets and Their Decision Problems“. In: *Transactions of the American Mathematical Society* 74.2 (Mar. 1953), pp. 358–366. ISSN: 0002-9947 (cit. on pp. 23, 112, 123).

[112] L. Rizzo. „Revisiting Network I/O APIs: The netmap Framework“. In: *ACM Queue* 10.1 (Jan. 2012), pp. 1–10. ISSN: 1542-7730 (cit. on p. 145).

[113] M. Roesch. „Snort - Lightweight Intrusion Detection for Networks“. In: *LISA '99: Proceedings of the 13th USENIX Conference on System Administration*. Seattle, Washington, USA, Nov. 1999, pp. 229–238 (cit. on p. 9).

[114] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy. „Exact Worst Case TCAM Rule Expansion“. In: *IEEE Transactions on Computers* 62.6 (June 2013), pp. 1127–1140. ISSN: 0018-9340 (cit. on pp. 15, 21, 107, 158, 174, 189).

[115] D. Rovniagin and A. Wool. „The Geometric Efficient Matching Algorithm for Firewalls“. In: *IEEE Transactions on Dependable and Secure Computing* 8.1 (Jan. 2011), pp. 147–159. ISSN: 1545-5971 (cit. on pp. 4, 16, 17, 21, 27, 50, 115).

[116] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous. „Survey and Taxonomy of IP Address Lookup Algorithms“. In: *IEEE Network: The Magazine of Global Internetworking* 15.2 (Mar. 2001), pp. 8–23. ISSN: 0890-8044 (cit. on p. 48).

[117] J. Salim, R. Olsson, and A. Kuznetsov. „Beyond Softnet“. In: *ALS 01: Proceedings of the 5th Annual Linux Showcase & Conference*. Oakland, California, USA, Nov. 2001, pp. 165–172 (cit. on p. 145).

[118] R. Sangireddy and A. Somani. „High-Speed IP Routing With Binary Decision Diagrams Based Hardware Address Lookup Engine“. In: *IEEE Journal on Selected Areas in Communications* 21.4 (May 2003), pp. 513–521. ISSN: 0733-8716 (cit. on p. 165).

[119] N. Sarnak and R. Tarjan. „Planar Point Location Using Persistent Search Trees“. In: *Communications of the ACM* 29.7 (July 1986), pp. 669–679. ISSN: 0001-0782 (cit. on p. 17).

[120] D. Shah and P. Gupta. „Fast Updating Algorithms for TCAMs“. In: *IEEE Micro* 21.1 (Jan. 2001), pp. 36–47. ISSN: 0272-1732 (cit. on pp. 79, 155).

[121] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. „PISCES: A Programmable, Protocol-Independent Software Switch“. In: *SIGCOMM '16: Proceedings of the 2016 ACM SIGCOMM Conference*. Florianopolis, Brazil, Aug. 2016, pp. 525–538 (cit. on p. 166).

[122] S. Singh, F. Baboescu, G. Varghese, and J. Wang. „Packet Classification Using Multidimensional Cutting“. In: *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Karlsruhe, Germany, Aug. 2003, pp. 213–224 (cit. on pp. 3, 4, 27–29, 50, 51, 53, 71, 103, 108, 120).

[123] R. Sinnappan and S. Hazelhurst. „A Reconfigurable Approach to Packet Filtering“. In: *FPL '01: Proceedings of the 11th International Workshop on Field-Programmable Logic and Applications*. Belfast, Northern Ireland, UK, Aug. 2001, pp. 638–642 (cit. on p. 165).

[124] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu. „DC.p4: Programming the Forwarding Plane of a Data-Center Switch“. In: *SOSR '15: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. Santa Clara, California, June 2015, pp. 1–8 (cit. on p. 166).

[125] H. Song and J. Lockwood. „Efficient Packet Classification for Network Intrusion Detection Using FPGA“. In: *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*. Monterey, California, USA, Feb. 2005, pp. 238–245 (cit. on pp. 6, 155, 176, 209).

[126] E. Spitznagel, D. Taylor, and J. Turner. „Packet Classification Using Extended TCAMs“. In: *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*. Atlanta, GA, USA, Nov. 2003, pp. 120–131 (cit. on pp. 155, 158).

[127] V. Srinivasan, S. Suri, and G. Varghese. „Packet Classification using Tuple Space Search“. In: *SIGCOMM '99: Proceedings of the 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Cambridge, MA, USA, Aug. 1999, pp. 135–146 (cit. on pp. 3, 5, 14, 15, 27, 29, 31, 33, 34, 36, 70, 79, 80, 95, 96).

[128] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. „Fast and Scalable Layer Four Switching“. In: *SIGCOMM '98: Proceedings of the 1998 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Vancouver, BC, Canada, Aug. 1998, pp. 191–202 (cit. on pp. 3, 27, 31, 42, 48, 70, 79).

[129] A. Tanenbaum, H. van Staveren, and J. Stevenson. „Using Peephole Optimization on Intermediate Code“. In: *ACM Transactions on Programming Languages and Systems* 4.1 (Jan. 1982), pp. 21–36. ISSN: 0164-0925 (cit. on p. 60).

[130] A. Tapdiya and E. Fulp. „Towards Optimal Firewall Rule Ordering Utilizing Directed Acyclical Graphs“. In: *ICCCN '09: Proceedings of 18th International Conference on Computer Communications and Networks*. San Francisco, CA, USA, Aug. 2009, pp. 1–6 (cit. on p. 117).

[131] D. Taylor. „Survey and Taxonomy of Packet Classification Techniques“. In: *ACM Computing Surveys* 37.3 (Sept. 2005), pp. 238–275. ISSN: 0360-0300 (cit. on p. 31).

[132] D. Taylor and J. Turner. „ClassBench: A Packet Classification Benchmark“. In: *IEEE/ACM Transactions on Networking* 15.3 (June 2007), pp. 499–511. ISSN: 1063-6692 (cit. on pp. 70, 96, 128, 142, 183, 204).

[133] R. Tessier and W. Burleson. „Reconfigurable Computing for Digital Signal Processing: A Survey“. In: *Journal of VLSI Signal Processing Systems* 28.1-2 (May 2001), pp. 7–27. ISSN: 0922-5773 (cit. on p. 151).

[134] Z. Ullah, M. Jaiswal, Y. Chan, and R. Cheung. „FPGA Implementation of SRAM-based Ternary Content Addressable Memory“. In: *IPDPSW '12: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, Workshops and PhD Forum*. Shanghai, China, May 2012, pp. 383–389 (cit. on p. 155).

[135] Z. Ullah, M. Jaiswal, and R. Cheung. „Z-TCAM: An SRAM-based Architecture for TCAM“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.2 (Feb. 2015), pp. 402–406. ISSN: 1063-8210 (cit. on p. 155).

[136] B. Vamanan and T. Vijaykumar. „TreeCAM: Decoupling Updates and Lookups in Packet Classification“. In: *CoNEXT '11: Proceedings of the Seventh Conference on Emerging Networking EXperiments and Technologies*. Tokyo, Japan, Dec. 2011, pp. 1–12 (cit. on pp. 79, 155).

[137] B. Vamanan, G. Voskuilen, and T. Vijaykumar. „EffiCuts: Optimizing Packet Classification for Memory and Throughput“. In: *SIGCOMM '10: Proceedings of the 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. New Delhi, India, Aug. 2010, pp. 207–218 (cit. on pp. 3, 4, 27, 29, 50, 71, 120, 145).

[138] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. „Multi-Layer Packet Classification with Graphics Processing Units“. In: *CoNEXT '14: Proceedings of the 10th International Conference on Emerging Networking Experiments and Technologies*. Sydney, Australia, Dec. 2014, pp. 109–120 (cit. on pp. 35, 79).

[139] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis. „Gnort: High Performance Network Intrusion Detection Using Graphics Processors“. In: *RAID '08: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*. Boston, MA, USA, Sept. 2008, pp. 116–134 (cit. on p. 3).

[140] N. Venugopal, V. Shobana, and R. Manimegalai. „Analysis of Optimization Techniques in FPGA Placement“. In: *ICCCI '14: Proceedings of the 2014 International Conference on Computer Communication and Informatics*. Coimbatore, India, Jan. 2014, pp. 1–5 (cit. on p. 153).

[141] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. „Scalable High Speed IP Routing Lookups“. In: *SIGCOMM '97: Proceedings of the 1997 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Cannes, France, Sept. 1997, pp. 25–36 (cit. on p. 48).

[142] H. Wang, R. Soulé, H. Dang, K. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. „P4FPGA: A Rapid Prototyping Framework for P4“. In: *SOSR '17: Proceedings of the 3rd ACM SIGCOMM Symposium on Software Defined Networking Research*. Santa Clara, CA, USA, Apr. 2017, pp. 122–135 (cit. on p. 166).

[143] P. Wang. „Scalable Packet Classification for Datacenter Networks“. In: *IEEE Journal on Selected Areas in Communications* 32.1 (Jan. 2014), pp. 124–137. ISSN: 0733-8716 (cit. on p. 145).

[144] P. Wang. „Scalable packet classification with controlled cross-producting“. In: *Computer Networks* 53.6 (Apr. 2009), pp. 821–834. ISSN: 1389-1286 (cit. on pp. 56, 127).

[145] M. Wegman and F. Zadeck. „Constant Propagation with Conditional Branches“. In: *ACM Transactions on Programming Languages and Systems* 13.2 (Apr. 1991), pp. 181–210. ISSN: 0164-0925 (cit. on pp. 60, 175).

[146] T. Woo. „A Modular Approach to Packet Classification: Algorithms and Results“. In: *INFOCOM '00: Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*. Tel Aviv, Israel, Mar. 2000, 1213–1222 vol.3 (cit. on pp. 3, 27, 50).

[147] B. Xu, D. Jiang, and J. Li. „HSM: A Fast Packet Classification Algorithm“. In: *AINA '05: Proceedings of the IEEE 19th International Conference on Advanced Information Networking and Applications*. New Taipei City, Taiwan, Mar. 2005, 987–992 vol.1 (cit. on pp. 27, 29, 47, 56, 127).

[148] F. Yu and R. Katz. „Efficient Multi-match Packet Classification with TCAM“. In: *HOTI 04: Proceedings of the 12th Annual Symposium on High-Performance Interconnects*. Stanford, CA, USA, Aug. 2004, pp. 28–34 (cit. on pp. 16, 156, 174).

[149] L. Yuan, H. Chen, J. Mai, C. Chuah, Z. Su, and P. Mohapatra. „FIREMAN: A Toolkit for FIREwall Modeling and ANalysis“. In: *S&P' 06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA, May 2006, pp. 199–213 (cit. on p. 16).

[150] C. Zerbini and J. Finochietto. „Performance Evaluation of Packet Classification on FPGA-based TCAM Emulation Architectures“. In: *GLOBECOM '12: Proceedings of the IEEE 2012 Global Communications Conference*. Anaheim, CA, USA, Dec. 2012, pp. 2766–2771 (cit. on pp. 155, 160).

[151] S. Zhou, S. Singapura, and V. Prasanna. „High-Performance Packet Classification on GPU“. In: *HPEC '14: Proceedings of the 2014 IEEE High Performance Extreme Computing Conference*. Waltham, MA, USA, Sept. 2014, pp. 1–6 (cit. on p. 39).

[152] D. Ziener, F. Bauer, A. Becher, C. Dennl, K. Meyer-Wegener, U. Schürfeld, J. Teich, J. Vogt, and H. Weber. „FPGA-Based Dynamically Reconfigurable SQL Query Processing“. In: *ACM Transactions on Reconfigurable Technology and Systems* 9.4 (Aug. 2016), 25:1–25:24. ISSN: 1936-7406 (cit. on p. 146).

Books

[153] C. Bobda. *Introduction to Reconfigurable Computing*. 1st ed. Springer, May 2007. ISBN: 978-1-4020-6088-5 (cit. on p. 152).

[154] R. Brayton, A. Sangiovanni-Vincentelli, C. McMullen, and G. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. 1st ed. Springer, Aug. 1984. ISBN: 0898381649 (cit. on pp. 153, 191).

[155] W. Cheswick, S. Bellovin, and A. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*. 2nd ed. Addison-Wesley Professional Computing Series, Feb. 2003. ISBN: 020163466X (cit. on p. 10).

[156] S. Churiwala. *Designing with Xilinx® FPGAs Using Vivado*. 1st ed. Springer, Nov. 2016. ISBN: 3319424378 (cit. on pp. 149–152).

[157] U. Farooq, Z. Marrakchi, and H. Mehrez. *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. 1st ed. Springer, May 2012. ISBN: 978-1-4614-3594-5 (cit. on pp. 151, 152).

[158] B. LaMeres. *Introduction to Logic Circuits & Logic Design with VHDL*. 1st ed. Springer, Oct. 2016. ISBN: 978-3-319-34194-1 (cit. on p. 152).

[159] M. Smith. *Application-Specific Integrated Circuits*. 1st ed. Addison-Wesley, June 1997. ISBN: 0201500221 (cit. on p. 146).

[160] W. Stallings. *Computer Organization & Architecture: Designing for Performance*. 6th ed. Pearson, July 2002. ISBN: 0130493074 (cit. on p. 149).

[161] A. Tanenbaum and T. Austin. *Structured Computer Organization*. 6th ed. Pearson, Aug. 2012. ISBN: 0273769243 (cit. on p. 149).

[162] S. Trimberger. *FIELD-PROGRAMMABLE GATE ARRAY TECHNOLOGY*. 1st ed. Springer, Jan. 1994. ISBN: 1461361834 (cit. on p. 146).

[163] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. 1st ed. Morgan Kaufmann Publishers Inc., Dec. 2004. ISBN: 0120884771 (cit. on p. 47).

[164] E. Zwicky, S. Cooper, and D. Chapman. *Building Internet Firewalls*. 2nd ed. O'Reilly & Associates, July 2000. ISBN: 1565928717 (cit. on p. 10).

Web Pages

[165] U. Antsilevich, P. Kamp, A. Nash, A. Cobbs, L. Rizzo, et al. *IPFW Firewall*. `www.freebsd.org/cgi/man.cgi?ipfw`. Last access: August 9, 2016 (cit. on pp. 4, 6, 32, 107, 120, 124, 158, 213).

[166] P. Ayuso, J. Kadlecsik, E. Leblond, F. Westphal, A. González, R. Russel, J. Morris, M. Boucher, H. Welte, M. Josefsson, Y. Kozakai, et al. *iptables-extensions*. `ipset.netfilter.org/iptables-extensions.man.html`. Last access: June 11, 2016 (cit. on pp. 4, 33, 60, 107, 123).

[167] P. Ayuso, J. Kadlecsik, E. Leblond, F. Westphal, A. González, R. Russel, J. Morris, M. Boucher, H. Welte, M. Josefsson, Y. Kozakai, et al. *The netfilter.org "iptables" project*. `www.netfilter.org/projects/iptables`. Last access: August 9, 2016 (cit. on pp. 3, 4, 6, 32, 107, 120, 158, 213).

[168] P. Ayuso, J. Kadlecsik, E. Leblond, F. Westphal, A. González, R. Russel, J. Morris, M. Boucher, H. Welte, M. Josefsson, Y. Kozakai, et al. *The netfilter.org "nftables" project*. `www.netfilter.org/projects/nftables`. Last access: August 9, 2016 (cit. on pp. 6, 120, 213).

[169] Intel® Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. `software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`. Last access: June 16, 2019 (cit. on p. 66).

[170] Intel® Corporation. *Intel® Intrinsics Guide*. `software.intel.com/sites/landingpage/IntrinsicsGuide`. Last access: June 16, 2019 (cit. on p. 66).

[171] M. Fleming. *A thorough introduction to eBPF*. `https://lwn.net/Articles/740157`. Last access: May 27, 2019 (cit. on p. 60).

[172] D. Hartmeier et al. *PF Firewall*. `www.openbsd.org/faq/pf`. Last access: August 9, 2016 (cit. on pp. 4, 16, 32, 158).

[173] V. Jacobson, C. Leres, S. McCanne, et al. *tcpdump*. `www.tcpdump.org`. Last access: May 25, 2019 (cit. on p. 60).

[174] A. Nygren, B. Pfaff, B. Lantz, B. Heller, C. Barker, C. Beckmann, D. Cohn, D. Malek, D. Talayco, D. Erickson, D. McDysan, D. Ward, E. Crabbe, F. Schneider, G. Gibb, G. Appenzeller, J. Tourrilhes, J. Tonsing, J. Pettit, K. Yap, L. Poutievski, L. Dunbar, L. Vicisano, M. Casado, M. Takahashi, M. Kobayashi, M. Orr, N. Yadav, N. McKeown, N. dHeureuse, P. Balland, R. Madabushi, R. Ramanathan, R. Price, R. Sherwood, S. Das, S. Gandham, S. Curtis, T. Mizrahi, T. Yabe, W. Ding, Y. Yiakoumis, Y. Moses, and Z. Kis. *OpenFlow Switch Specification Version 1.5.0*. `www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf`. Last access: Nov. 26, 2017 (cit. on p. 139).

[175] D. Reed et al. *IPFILTER*. `www.freebsd.org/doc/handbook/firewalls-ipf.html`. Last access: June 11, 2016 (cit. on pp. 32, 107, 140).

[176] A. Turner et al. *Tcpreplay*. `tcpreplay.synfin.net`. Last access: November 24, 2017 (cit. on p. 129).

Standards

[177] S. Kent and R. Atkinson. *RFC 2406: IP Encapsulating Security Payload (ESP)*. Tech. rep. Nov. 1998, pp. 1–22 (cit. on p. 17).

[178] J. Mogul and S. Deering. *RFC 1191: Path MTU Discovery*. Tech. rep. Nov. 1990, pp. 1–19 (cit. on p. 11).

[179] J. Postel. *RFC 791: Internet Protocol*. Tech. rep. Sept. 1981, pp. 1–51 (cit. on p. 12).

[180] K. Varadhan. *RFC 1519: Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*. Tech. rep. Sept. 1993, pp. 1–24 (cit. on p. 157).

White Papers

[181] Altera. *FPGA Architecture*. `https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf`. Last access: Apr. 29, 2018 (cit. on pp. 150, 151).

[182] A. Goldhammer and J. Ayer. *Understanding Performance of PCI Express Systems*. `https://www.xilinx.com/support/documentation/white_papers/wp350.pdf`. Last access: Apr. 15, 2019 (cit. on p. 207).

Selbständigkeitserklärung / Declaration

Hiermit erkläre ich, die Dissertation selbstständig und nur unter Verwendung der angegebenen Hilfen und Hilfsmittel angefertigt zu haben. Ich habe mich nicht anderwärts um einen Doktorgrad in dem Promotionsfach beworben und besitze keinen entsprechenden Doktorgrad. Die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 34/2006 am 03.08.2006, habe ich zur Kenntnis genommen.

I declare that I have completed the thesis independently using only the aids and tools specified. I have not applied for a doctor's degree in the doctoral subject elsewhere and do not hold a corresponding doctor's degree. I have taken due note of the Faculty of Mathematics and Natural Sciences PhD Regulations, published in the Official Gazette of Humboldt-Universität zu Berlin no. 34/2006 on 03/08/2006.

Berlin, 16.01.2020

Sven Hager